NCS TIB 95-4

NATIONAL COMMUNICATIONS SYSTEM

TECHNICAL INFORMATION BULLETIN 95-4

TAMI MODEL PROGRAMMER'S GUIDE VOLUME II

JUNE 1995

OFFICE OF THE MANAGER
NATIONAL COMMUNICATIONS SYSTEM
701 SOUTH COURT HOUSE ROAD
ARLINGTON, VA 22204-2198

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	Final Report

4. TITLE AND SUBTITLE

TAMI Model Programmer's Guide Volume II

5. FUNDING NUMBERS

DCA100-91-C-0015

6. AUTHOR(S)

Andre Rausch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Booz, Allen & Hamilton, Inc.
8283 Greensboro Drive
McLean, Virginia 22102

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Communications System
Office of Technology and Standards Division
701 South Court House Road
Arlington, Virginia 22204-2198

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NCS TIB #95-4

VOL II

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As part of the ongoing effort to analyze the performance of the public switched network (PSN) and the programs of the National Level National Security and Emergency Preparedness (NS/EP) Telecommunications Program (NLP), Office of Technology and Standards Division (N6) has developed a number of computer-based models. The Traffic Analysis by Method of Iteration (TAMI) model was developed to measure the effects of telecommunications traffic congestion in stressed local and long distance networks. This document provides the second of two volumes of the TAMI Programmer's Manual. These volumes provide the software description necessary for a programmer to support future maintenance and enhancements to the TAMI model. This volume documents the remaining 12 modules. It is assumed that the reader has a basic understanding of the PSN and a working knowledge of the architectures of the three major inter-exchange carrier networks (IEC) and the local exchange carrier networks (LEC). A programmer using TAMI should have a working knowledge of the UNIX operating system and the 'C' and FORTRAN programming languages. A background in voice teletraffic engineering and analytical modeling and simulation is desirable.

14. SUBJECT TERMS

TAMI Modules, Public Switched Network (PSN)
Inter-Exchange Carrier (IEC) Networks
Local Exchange Carrier (LEC) Networks

15. NUMBER OF PAGES

230

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASS	UNCLASS	UNCLASS	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

NCS TIB 95-4

NATIONAL COMMUNICATIONS SYSTEM

TECHNICAL INFORMATION BULLETIN 95-4

TAMI MODEL PROGRAMMER'S GUIDE VOLUME II

JUNE 1995

OFFICE OF THE MANAGER
NATIONAL COMMUNICATIONS SYSTEM
701 SOUTH COURT HOUSE ROAD
ARLINGTON, VA 22204-2198

NCS TECHNICAL INFORMATION BULLETIN 95-4

TAMI MODEL PROGRAMMER'S GUIDE VOLUME II

JUNE 1995

PROJECT OFFICER

ANDRE RAUSCH
Electronics Engineer
Office of Technology
and Standards

APPROVED FOR PUBLICATION:

DENNIS BODSON
Assistant Manager
Office of Technology
and Standards

FOREWORD

The National Communications System (NCS) is an organization of the Federal Government whose membership is comprised of 23 Government entities. Its mission is to assist the President, National Security Council, Office of Science and Technology Policy, and Office of Management and Budget in:

- The exercise of their wartime and non-wartime emergency functions and their planning and oversight responsibilities.
- The coordination of the planning for and provisions of National Security/Emergency Preparedness communications for the Federal Government under all circumstances including crisis or emergency.

In support of this mission, the NCS has conducted studies and analyses to assess the potential for serious damage to portions of the Nation's telecommunications infrastructure due to various threats. The purpose of the work is to provide guidance to programmers on the TAMI module structure.

Comments on this TIB are welcome and should be addressed to:

Office of the Manager
National Communications System
Attn: NC-TS
701 S. Court House Road
Arlington, VA 22204-2198

Table Of Contents

TAMI Model Programmer's Guide, Part II

1.0 Introduction

The Office of the Manager, National Communications System (OMNCS) Office of Technology and Standards (NT) is responsible for a broad range of initiatives including Federal telecommunications standards development, network performance analyses, and technology review. As part of the ongoing effort to analyze the performance of the public switched network (PSN) and the programs of the National Level National Security and Emergency Preparedness (NS/EP) Telecommunications Program (NLP), NT has developed a number of computer-based models. Most recently, the Traffic Analysis by Method of Iteration (TAMI) model was developed to measure the effects of telecommunications traffic congestion in stressed local and long distance networks.

1.1 Purpose

This document provides the second of two volumes of the TAMI Programmer's Manual. Together, these volumes provide the software description necessary for a programmer to support future maintenance and enhancements to the TAMI model.

1.2 Scope

Volume I of the TAMI Programmer's Manual documents the first 11 of 23 modules that form the TAMI model. Volume II of the TAMI Programmer's Manual, this volume, documents the remaining 12 modules. It is assumed that the reader has a basic understanding of the PSN and a working knowledge of the architectures of the three major inter-exchange carrier networks (IEC) and the local exchange carrier networks (LEC). Furthermore, a programmer using TAMI should have a working knowledge of the UNIX operating system and the 'C' and FORTRAN programming languages. A background in voice teletraffic engineering and analytical modeling and simulation is desirable to understand the algorithmic details of the TAMI model. The TAMI programmer will find it useful to be familiar with the references provided at the end of this document, which describe previous TAMI analyses, modeling concepts, algorithms, and programmer's manuals of related software.

1.3 Background

The nation's PSNs continue to be a focus of NCS modeling efforts because these networks comprise the largest, most diverse set of telecommunications assets in the United States. Furthermore, the NCS directs its NS/EP telecommunications enhancement activities toward the PSN. Additionally, most NCS member organizations rely on the PSN for conducting their NS/EP responsibilities.

The NCS has moved to measuring network performance using call completion probability in addition to connectivity because this approach captures the effects of traffic congestion. Traffic congestion is prevalent during many of the national emergencies and disasters of concern to the OMNCS.

In support of PSN traffic congestion analyses, NT has developed the TAMI model. This model measures congestion in the combined local and long-distance networks of the PSN. TAMI evaluates congestion for ordinary telephone users and for NS/EP users who benefit from planned or existing NLP enhancements. In addition to measuring nationwide congestion, the TAMI model has been expanded to model regional congestion caused by focused overloads. Focused overloads are common during events that only affect part of the country, such as earthquakes and hurricanes, during which the affected region may be subject to unusually high volumes of traffic originating from the rest of the country. As the TAMI model continues to evolve, it provides a more accurate tool for understanding the effects of congestion in the PSN.

The TAMI model has been used by both NT and the Office of Plans and Programs (NP) to measure congestion in the PSN subject to damage from electromagnetic pulse (EMP), fallout radiation, nuclear blast scenarios, and, more recently, earthquakes. An analysis has successfully been conducted to determine the sensitivity of the model's network performance results to network management and engineering assumptions made in the absence of complete PSN data. In view of continued plans to employ and enhance the TAMI model, this document provides the second of two Programmer's Manual volumes.

1.4 Organization

This report is organized into three sections. Section 1.0, this section, provides an introduction, describing the purpose, scope, background, and organization, as well as a short description of documentation conventions.

Section 2.0 provides a high level overview of the TAMI model and describes the interrelationships, dataflow, and interfaces among the twelve software modules encompassed by this report.

Section 3.0 contains detailed documentation of the remaining 12 TAMI software modules and each module's component functions.

1.5 Document Conventions

This manual documents modules and functions coded in the 'C' programming language. Throughout the manual certain conventions which may differ slightly from standard 'C' terminology have been adopted in order to more clearly describe data types, inputs, outputs and file types. In addition the typographic `courier` font is used to denote variable names.

Variable names are defined by the following conventions:

Convention	Example	Definition
character	`c`	**The standard 'C' data type *char***
integer	`i`	**The standard 'C' data type *int***
float	`real`	**The standard 'C' data type *float***
double	`long_real`	**The standard 'C' data type *double***
long integer	`pos`	**The standard 'C' data type *long***
file	`outfile`	**The standard 'C' data type *FILE*, a pointer to a file string**
extern	`optarg`	**The standard 'C' data type modifier *extern* indicating the variable is declared outside the module (e.g., the operating system)**
global	`idx_num`	**Variables that are declared globally accessible from any function**
pointer	`*varname`	**Denotes a pointer to any variable, *varname***
structure	`p_struct p[]` *with field* integer `p[].three`	**The standard 'C' aggregate, heterogeneous hierarchical data structure composed of a main variable name and sub fields of multiple data types**

Inputs/Outputs are defined by the following conventions:
Inputs are of two type: 1) formal inputs are passed in by the calling function; 2) global inputs are variables as defined above. Outputs are of two type: 1) formal outputs are passed out by the called function as standard 'C' returns; 2) global outputs are variables as defined above

Includes are defined by the following conventions:
Includes are of two types: 1) Standard 'C' defined function sets, e.g., `<stdio.c>`; and 2) user defined function sets, e.g., `"fileio.c"` (see Appendix A)

File formats are defined by the following conventions:

1) <CLLIA>, <CLLIZ>, <size>
2) (c11, 1x, c11, 1x, i6)

Line 1 shows the names and relative positions of the fields within each record. Line 2 shows the data type and length of each field, where:

c=character
x=space
i=integer
f=float

2.0 High Level TAMI Description

This section provides a high level overview of the TAMI Model from a software viewpoint. A number of NCS reports already exist which describe the TAMI algorithms, assumptions, and modeling techniques (References 3, 5, 6, 7). The purpose of this overview is to focus on the interrelationships, data flow, and interfaces among the software modules that constitute the TAMI model, focusing on those that were not already covered in Volume I.

The TAMI model can be divided into six main functional processes, depicted in Exhibit 2.1. Each of these processes operate on both IEC and LEC data and can be described at more detailed levels. A brief summary of the six steps is provided here:

Step 1, Pre-Process Undamaged Network and Traffic, involves developing a complete picture of the undamaged LEC and IEC networks, including physical transmission paths, sizes of logical trunk groups, and primary and alternate routing considerations. This step then involves using telco engineering methodologies to engineer peak-hour traffic over each trunk group, reverse-engineer the sizes of certain categories of trunk groups, and determine the overall end-to-end traffic.

Step 2, Generate Pool of Damage Vectors, involves deriving several random damage scenarios from a probabilistic description of a network threat. Although the original damage module used with TAMI generated EMP and Fallout Radiation damage, any module may be substituted here to derive damage for other threats such as earthquakes or hurricanes. The important point is that each version of this step performs the same basic task of translating probabilistic damage into a large pool of random possible outcomes (damage vectors) of telecommunications network damage. The Monte Carlo sampling loop can later sample these damage vectors one at a time until an average is reached within the required confidence interval.

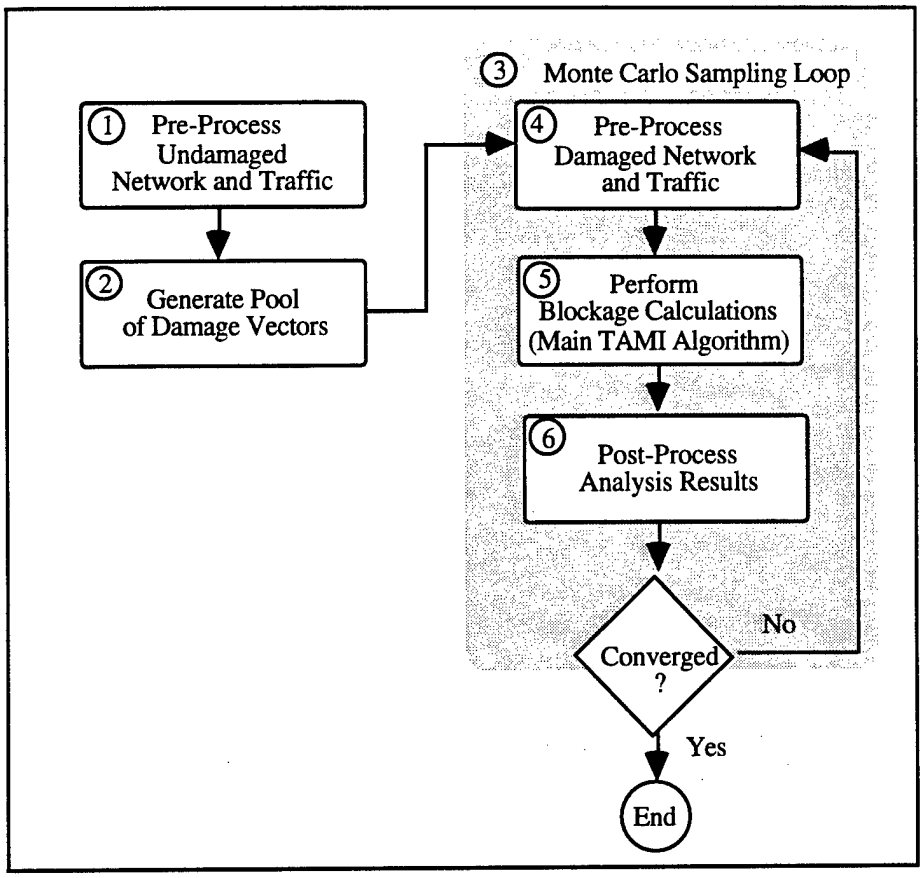
Step 3, Monte Carlo Sampling Loop, is the control loop for steps 4, 5, and 6. While these later steps each operate on only one damage vector at a time, the controlling logic in step 3 is responsible for selecting these damage vectors from the sampling pool, issuing the commands to run subsequent steps, and comparing running Monte Carlo statistics to the target confidence interval. If the target has not been reached, the Monte Carlo Sampling Loop repeats steps 4, 5, and 6 with the next damage vector in the sampling pool.

Step 4, Pre-Process Damaged Network and Traffic, involves applying the effects of a particular damage vector to the undamaged network. Trunk group sizes are set to zero where completely damaged, or adjusted appropriately where only some of the trunk group's diverse physical routes are damaged. The effects of damage on offered traffic and traffic distribution patterns are also calculated. The end result of Step 4 is a representation of both the damaged network and a damaged traffic matrix.

Step 5, Perform Blockage Calculations (Main TAMI Algorithm), is the first point where network congestion is measured for a given damage vector, using the outputs from Step 4. The post-damage traffic matrix is offered to the damaged network, and blockages are measured in each trunk group in both the LEC and IEC networks. When all the networks reach steady state, the individual trunk group blockages are combined to estimate end-to-end blockage through the PSN for that damage vector.

Step 6, Post-Process Analysis Results, is responsible for keeping the cumulative performance statistics for all damage vectors sampled so far. Therefore, every time Step 5 finishes, this step incorporates the results into its running average, increments the number of damage vectors sampled, and recomputes statistics such as variance. These statistics are examined by the controlling logic in Step 3 to determine convergence.

Exhibit 2.1
TAMI High-Level Flow Diagram



In addition to QTCM, which has been previously documented as a stand-alone model, there are 23 TAMI modules totaling an estimated 30,000 lines of code. Exhibit 2.2 categorizes each of these modules into the above six steps. It also provides the approximate lines of code for each module and indicates whether it is documented in Volume I or Volume II of the TAMI Model Programmer's Manual.

Exhibit 2.2
Table of TAMI Modules

Step #	Functional Area	Module Name	Approximate Lines of Code	Vol I	Vol II
1.0	Pre-Process Undamaged Network and Traffic				
1.1	IEC Networks	cg_make	200	3	
		span_make	300	3	
		array_make	200	3	
		mk_ncam_path	4800*	3	
		rem_dup	200	3	
		sort_path	130	3	
		3ch_4ch	200	3	
		match_trunk	830	3	
		mkpath	275	3	
1.2	LEC Networks and	attlive	900		3
	End-to-End Traffic Matrix	mcilive	1300		3
		sprlive	1300		3
2.0	Generate Pool of Damage Vectors	damage	1600	3	
		mklink	500	3	
3.0	Monte Carlo Sampling Loop	tami	650		3
4.0	Pre-Process Damaged Network and Traffic				
4.1	IEC Networks	mkrout	700		3
		qtrans_gen	800		3
4.2	LEC Networks and	attwdmg	2300		3
	End-to-End Traffic Matrix	mciwdmg	2300		3
		sprwdmg	2300		3
		merge	850		3
5.0	Perform Blockage Calculations				
5.1	LEC Networks	lecam	4500		3
5.2	IEC Networks (QTCM)	qmod_att	N/A	N/A	N/A
		qmod_mci	N/A	N/A	N/A
		qmod_spr	N/A	N/A	N/A
6.0	Post-Process Analysis Results	keepstats	300		3

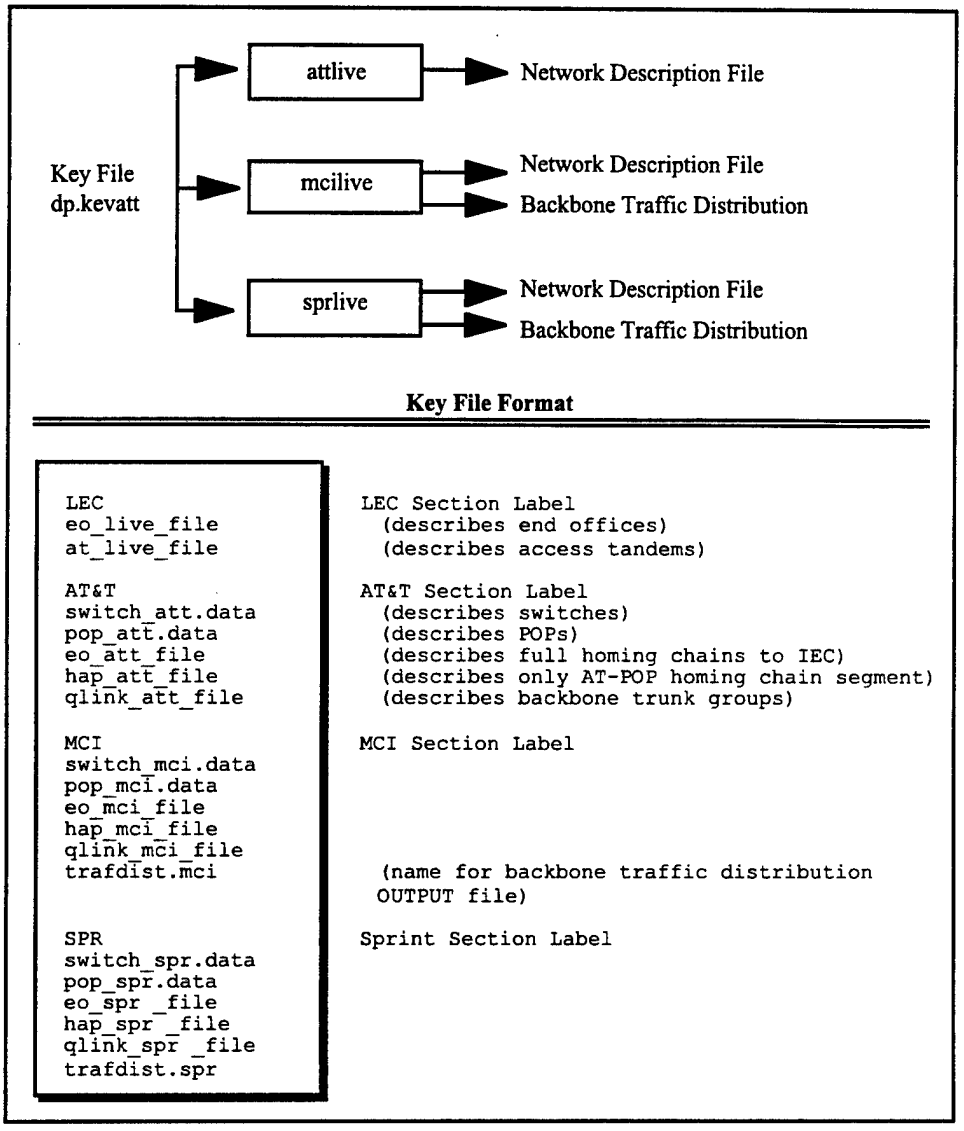
*Includes code linked from IDA/CAM model, Reference 5

As shown, Volume I encompasses Step 1.1: the pre-processing of the undamaged IEC networks and Step 2: the generation of sampling pools of EMP damage vectors. Volume II encompasses the remaining areas. While the detailed documentation for each module appears in Section 3, the remainder of this section is dedicated to describing the data flow among these modules and providing a brief description of each module, using the following subsections:

2.1 (Step 1.2) Pre-Process Undamaged LEC Networks and Traffic Matrix
2.2 (Step 3) Monte Carlo Sampling Loop
2.3 (Step 4.1) Pre-Process Damaged IEC Networks
2.4 (Step 4.2) Pre-Process Damaged LEC Networks and Traffic Matrix
2.5 (Step 5.1) Perform LEC Blockage Calculations
2.6 (Step 5.2) Perform IEC Blockage Calculations
2.7 (Step 6) Post-Process Analysis Results

Exhibits 2.3 through 2.5 provide a diagrammatic road map to these sections, depicting the overall data flow for AT&T, MCI, and Sprint network data through the Volume II modules. Exhibit 2.3 describes Step 1.2. Note that, while there is a different module for each IEC, the list of input files for all three modules is stored in a common 'dp.key' file. As shown, this file has a section for LEC files common to all three IECs and separate sections for each IEC. Each module uses only the files in the pertinent sections.

Exhibit 2.3
Dataflow Through Step 1.2



Because the dataflow for Step 3 and the steps controlled by it (Steps 4, 5, and 6) is so extensive, it has been divided into two diagrams, Exhibits 2.4 and 2.5. Exhibit 2.4 shows the dataflow for Steps 4.1 and 4.2, while exhibit 2.5 describes the dataflow for Steps 5 and 6. Note that Step 3 uses a key file 'tami.key.' This key file has a similar structure to the key file used in Step 1.2, including a section for the LEC and each IEC; however, some header flags have been added and there is a different set of file names in each section, including optional files and damage files. Step 3 will pass the appropriate command-line options and files from 'tami.key' to the steps that it controls.

Exhibit 2.4
Dataflow Through Steps 4.1 and 4.2

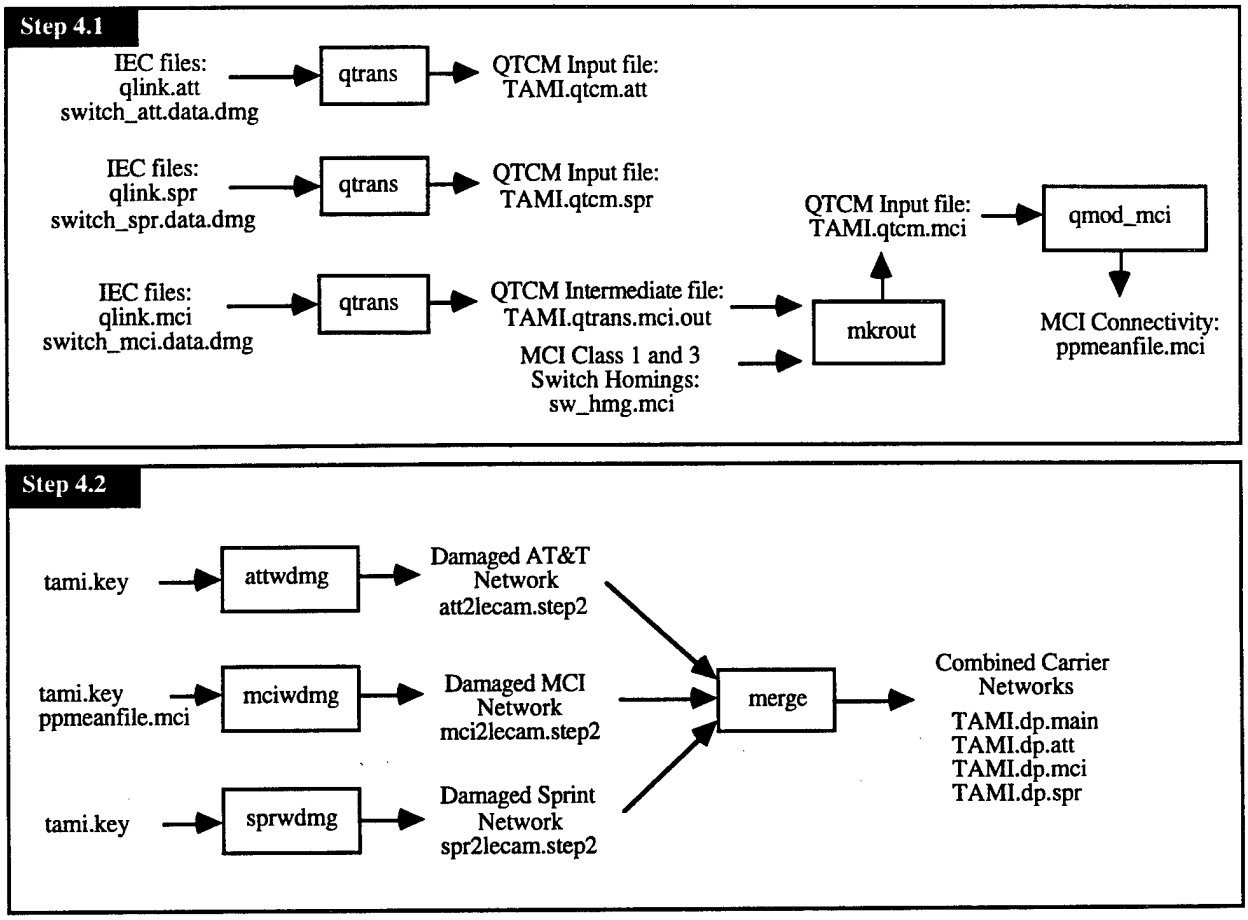
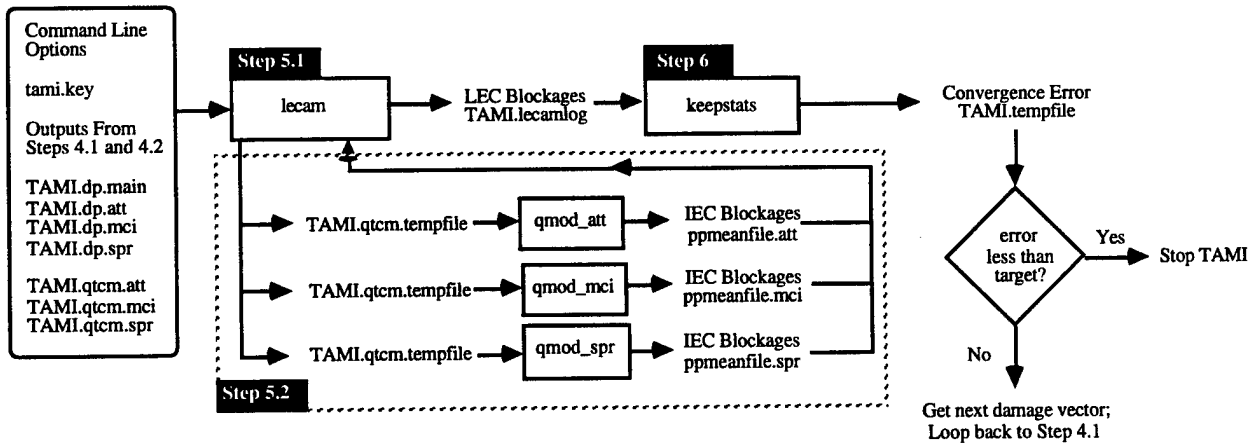


Exhibit 2.5
Dataflow Through Steps 5 and 6



2.1 (Step 1.2) Pre-Process Undamaged LEC Networks and Traffic Matrix

There are five goals to this step: (1) derive access traffic at the end offices, (2) derive the engineered traffic on all LEC trunk groups, (3) reverse engineer the sizes of the AT-POP and EO-POP trunk groups (AT&T only), (4) determine whether the transmission paths for high usage trunk groups are physically diverse from the AT, (5) determine the backbone traffic distribution for each IEC. This section describes the modules in this step, and concludes with a brief discussion of the importance of appropriately representing the traffic matrices in TAMI. All the computations and algorithms used in Step 1.2 are described more thoroughly in a number of TAMI references: (7, 3).

To adapt to differences in each carrier's data and modeling assumptions, there is a dedicated module for each IEC: *attlive, mcilive, sprlive.* These modules are referred to as the "undamaged data preparation modules." Because the modules require a number of input files, a common "keyfile" concept was developed. This file contains labeled sections for each carrier, which lists all the files specific to that carrier. A general LEC section lists file names common to all three carriers. The key file is supplied to each module, which then scans the appropriate sections for the names of its input files. The keyfile format is depicted in Appendix B.

The data preparation modules compute a number of variables required at later stages. To simplify the handling of output files, these variables are incorporated into a single, multi-section output file. Each section is identified by a unique label. The exception is the backbone traffic distribution, which is stored in its own file by convention.

Engineered traffic numbers are computed in the data preparation modules at every possible point through the networks, including:

- access traffic at the end offices
- traffic engineered to be carried on the high-usage trunk group
- traffic engineered to be carried on the EO-AT trunk group
- aggregate traffic engineered to be carried from an AT to the IEC POP
- total traffic engineered to arrive at the IEC switch.

In addition, the backbone traffic distribution is computed to determine the proportion of traffic arriving at an IEC switch that gets distributed to each destination IEC switch.

Specific to *attlive,* sizes of EO-POP and AT-POP trunk groups are calculated. Unlike MCI and Sprint data, AT&T data does not indicate the sizes of these trunk groups connecting the LEC networks to its toll network. Trunk group sizes are engineered using the engineered traffic and telco standard grade of services targets.

Because physical transmission paths are not provided in the LEC networks, a physical diversity model is applied to all high usage trunk groups. This model determines whether the high usage trunk groups are likely to follow a physical path through the site of the local access tandem, or through a route that is diverse from the access tandem. This results determines whether high usage trunk groups are knocked out when there is damage to the local access tandem facility.

TAMI Traffic Matrices

The backbone traffic distribution is of particular importance in this stage. To illustrate this, it is necessary to explain how TAMI stores its "traffic matrix." In the abstract, the traffic matrix describes how much traffic is offered from EO i to EO j through IEC c:

$$T_{ij,c}$$

However, deriving this structure requires a large amount of computing time, and storing it provides an even greater barrier. Given approximately 15,000 end offices, Tij,c would require 675 million double precision floating point numbers, or about 5.4 gigabytes of disk storage. TAMI addresses this issue by aggregating traffic where possible, only distributing it at network branching points. Consider ACk,c, which is the total traffic offered by EO k to IEC c. This variable requires the storage of only k*c values. ACk,c will travel through the LEC network until it arrives

at an IEC switch. At this point, it must be distributed from switch Si,c in IEC c to all the other switches in the backbone (Sj,c), using a distribution matrix:

Sij,c

Because there are many fewer long distance toll switches than there are end offices, Sij,c is a relatively small structure that can be computed and used to distribute traffic through the toll networks. Traffic terminating at a switch, Sj,c, is the aggregate of traffic from all source switches, and must be distributed to the destination end offices to complete. Here, it is assumed that traffic terminates to end offices in the same proportions by which it originates. Understanding these traffic matrix calculations is essential for understanding many of the computations and variables required in Step 1.2 and in subsequent steps. The traffic matrix computations are presented in more detail in Reference 7 (Network Congestion Analysis).

The undamaged data preparation modules compute the access traffic (ACk,c) for each EO, and the backbone traffic distribution matrix, SSij,c. Together, these are used to describe the "end-to-end traffic matrix," Tij,c, required by TAMI.

2.2 (Step 3) Monte Carlo Sampling Loop

The purpose of this step is to run steps 4 through 6 for a single damage vector at a time, sampling as many damage vectors as required for results to converge. The Monte Carlo sampling process is described in more detail in Reference 7.

This step consists of a single software module, `tami`. Because `tami` controls steps 4 through 6, it is often referred to as the "controlling logic" for the TAMI model. `Tami` supports all of the command-line options of the modules it controls, which are passed down to these modules when they are called. In addition, `tami` reads and interprets the keyfile, and passes file names from it to called modules.

The main purpose of `tami` is to control the damage vector sampling process. It begins by running steps 4 through 6 on the first damage vector in the damage files. For the data flow depicted in Exhibits 2-2 and 2-3, it invokes all modules in the appropriate order, builds any necessary input files for each module, and passes appropriate command line arguments to each module. It then uses the error output by keepstats to decide if it needs to run steps 4 through 6 again using the next damage vector. The damage vector sampling loop continues until convergence is reached.

This step also coordinates multiple damage vector bins, as defined in the `tami` keyfile, if multiple damage scenarios are included in the same damage files. A set of cumulative results files are maintained for each bin. As `tami` runs, it prints ongoing status to the screen, such as the bin and damage vector currently being sampled, the module currently being run, and start and end time stamps for each loop.

2.3 (Step 4.1) Pre-Process Damaged IEC Networks

The goal of this step is to derive QTCM input files for each IEC, incorporating damage from the current sampled damage vector. In addition, for MCI, additional steps are performed to test backbone connectivity under the current damage vector.

Step 4.1 consists of three modules: `qtrans`, `mkrout` (MCI only) and `qmod_mci` (MCI only). `Qtrans` translates the qlink and switch files into QTCM file format. Because MCI uses hierarchical routing, `mkrout` is also required in order to derive a point-to-point routing table, which is incorporated into the QTCM file format.

Finally, `qmod_mci` is run on the MCI QTCM file using a very small offered load as a "trickle current." The resulting output blockage matrix, ppmeanfile.mci, will indicate whether at least one undamaged direct or indirect route exists between each switch pair through the hierarchical backbone. This information is required by the `mciwdmg` module to determine connectivity through the IEC.

2.4 (Step 4.2) Pre-Process Damaged LEC Networks and Traffic Matrix

This goal of this step is to incorporate the current damage vector into the LEC networks (EO to IEC switch) and traffic matrix offered to the PSN. This step uses the engineered (undamaged) network and traffic descriptions computed in step 1.2 as a baseline for applying damage. The output from this step is an input to the lecam module in Step 5.1.

Step 4.2 consists of four modules: `attwdmg`, `mciwdmg`, `sprwdmg`, and `merge`. Each of the "wdmg" modules handles the specific details of applying damage to the network and calculating the traffic matrix. These modules results in output network files for each IEC. `Merge` combines the sections of these files that are common to all three carriers. It also engineers portions of the LEC which require knowing the combined traffic from all IECs.

2.5 (Step 5.1) Perform LEC Blockage Calculations

There purpose of this step is to perform the blockage calculations in the LEC networks. Combined with the IEC blockages from Step 5.2, this step calculates overall PSN blockage statistics for the current sampled damage vector.

This step consists of a single module, `lecam`. The primary functions of `lecam` are as follows:

(1) calculate LEC blockages
(2) pass traffic up to IECs (QTCM) for IEC blockage calculations
(3) examine all (LEC and IEC) blockage calculations to determine convergence
(4) upon convergence, use EO pair file to calculate end-to-end blockage through each IEC
(5) separately calculate NS/EP blockages, dependent on NS/EP options (e.g., LEC TCR, E-RTNR, etc.)
(6) write these results to an output file

The output from `lecam` is used by the `keepstats` module in Step 6.

2.6 (Step 5.2) Perform IEC Blockage Calculations

The purpose of this step is to calculate the blockage through each IEC based on traffic offered from Step 5.1. The execution of this step is used as a subroutine to Step 5.1, which controls its execution.

This step consists of the QTCM model. Separate versions (`qmod_att`, `qmod_mci`, `qmod_spr`) have been created for each IEC since input and output filenames are hardcoded for each carrier. The purpose of QTCM is to calculate the point-to-point blockages in each IEC backbone network. These results are written to the 'ppmeanfile' output files which are interpreted by Step 5.1.

2.7 (Step 6) Post-Process Analysis Results

The purpose of this step is to maintain cumulative statistics for the Monte Carlo sampling process. Each time PSN performance results are determined for a given damage vector, this step accumulates them into the running averages and totals and calculates the "error" in the sampling process.

Step 6 consists of a single module, `keepstats`. The error calculated by `keepstats` is used by the controlling logic in Step 3 to determine if another damage vector should be sampled or if results have converged.

3.0 MODULE DESCRIPTIONS

3.1 attlive: Build AT&T Live Network and Traffic

Purpose

The purpose of this module is to complete the representation of the undamaged AT&T network. This module uses existing publicly available data on that network along with telco engineering and reverse engineering methods to completely represent the AT&T network from the TGs down to the LECs, and to build a traffic matrix to populate that network. This module is used in conjunction with `attwdmg` which introduces damage effects into the network data files. The final output of the two step process after the `attwdmg` module is input into the LECAM module.

Call Syntax

`attlive -k <key file> [ options]`

options:	*special syntax:*	*function:*
`-k`	`<keyfile>`	reads in input file `<key file>`
`-o`	`<outfile>`	reads in output file name. If not specified, default file name is `att2lecam.step1`
`-t`	`<#>`	switch index for backbone test, (-1) for all switches
`-h`		invokes new High Usage occupancy and overflow algorithm commands (this setting is now considered the default setting)
`-?`		user help--prints call syntax and exits without running

example `attlive -k dplive.key -h` (spaces optional)

Input Files

key fileThis file contains hardcoded instructions for opening up and reading in the various data files to be used in this module.

format **SEE SECTION 2.1/EXHIBIT 2-3**

Output Files

att2lecam file This file contains the formatted output data to be used in the LECAM module. It contains a header section followed by 17 data sections.

format
header:

* ATT live data file (damage added in MC loop)
<IEC label>
("IECid=", c3)
<switch size>
("SWITCH_SIZE", i3)
<POP size>
("POP_SIZE", i3)
<AT size>
("AT_SIZE", i4)
<EO size>
("EO_SIZE", i5)
<AT to POP size>
("ATPOPSIZE", i3)

EOAT section 1:	<traffic at the AT for each EO>, (i4, I4,.....) in rows of 20
HUPOP section 2:	<traffic between high usage AT and the POP for each EO>, (i4, I4,.....) in rows of 20
ATPOP section 3:	<traffic between AT and the POP for each EO>, (i4, I4,.....) in rows of 20
HAP section 4:	<number of high usage trunks AT and the POP for each EO>, (i4, I4,.....) in rows of 20
SDlive section 5:	< size of trunks between EO and POP>, (i5, I5,.....) in rows of 15
TEP section 6:	<traffic between EO and the POP for each EO>, (f8.2, f8.2, ...) in rows of 10
TAP section 7:	<traffic between AT and the POP for each EO>, (f8.2, f8.2, ...) in rows of 10
F0 section 8:	<number of high usage trunks AT and the POP for each EO>, (f14.8, f14.8, ...) in rows of 2
AClive section 9:	<total access capacity>, (f.8, f.8, ...) in rows of 5
ACSlive section 10:	< total access capacity for the LEC>, (f.8, f.8, ...) in rows of 5
ELAP section 11:	<traffic between AT and the POP for each AT>, (f8.2, f8.2, ...) in rows of 6
SAPlive section 12:	<size of trunks between AT and the POP for each AT>, (i5, i5,.....) in rows of 6
D3 section 13:	< access traffic engineered from LEC switches to POP>, (f8.2, f8.2, ...) in rows of 10
ACSPOP section 14:	< access traffic engineered from IEC switches to POP>, (f8.2, f8.2, ...) in rows of 10
POPSWITCH section 15:	< POP switch homing>, (i4, i4, ...) in rows of 20
EOlata section 16:	< LATA each EO is homed to >, (i4, i4, i4, ...) in rows of 20
PhysDiv section 17:	<physical diversity value for each end office: 0/1>, (i1, i1, ...) in rows of 80
IMT section 18:	< matrix of backbone trunk traffic for each switch>, (i5, i5, ...) in rows of 15
PropIntra section 19:	< proportion of intra-switch traffic for each switch>, (f.8, f.8, ...) in rows of 5

Includes

"attlive.h"	Defines global network and traffic variables and constants.
"lecam.h"	Defines network sizing constants.
"fileio.c"	User-defined I/O functions; see Appendix A.

Constants

Constants used in attlive and defined in "attlive.h":

OVFLW	0.90	AT&T overflows 10% from HU to final
TRKOCC	0.95	HUs are 95% occupied
INTRA	0.0	multiplier for inter-switch traffic to add intra-switch traffic

Constants used in attlive and defined in "lecam.h":

MAX_AT	1000	maximum number of access tandems
MAX_EO	19250	maximum number of end offices
MAX_POP	575	maximum number of points of presence
MAX_SWITCH	130	maximum number of switches

Global Variables

global variables defined in "attlive.h"

integer	KEYTOG	toggle to indicate input 'file name' was read in successfully
	NUMEO	number of end offices
	NUMAT	number of access tandems
	NUMPOP	number of points of presence
	NUMSW	number of switches
	ATPOPSIZE	the number of trunks between the AT and the POP
	HAP[MAX_AT] [ATPOPMAX]	number of high usage trunks between the AT and the POP
	AT[MAX_EO]	traffic at the AT
	ATPOP[MAX_EO]	traffic between the AT and the POP
	HUPOP[MAX_EO]	traffic between the high usage AT and the POP
	TYPE[MAX_EO]	type of homing between the AT and the SW
	SW[MAX_EO]	traffic at the switch
	LATA[MAX_EO]	traffic at the LATA
	COC[MAX_EO]	central office code homings
	IMT[MAX_SWITCH] [MAX_SWITCH]	matrix of backbone trunk traffic
	SEP[MAX_EO]	size of trunks between EO and POP
	SAP[MAX_AT] [ATPOPMAX]	size of trunks between AT and POP
	COCSW[MAX_SWITCH]	matrix of homings between switches and COCs
	PhysDiv[MAX_EO]	physical diversity value for each EO of TYPE = 3
	SW_nlata[MAX_SWITCH]	number of LATAs that a switch serves
	popswitch[MAX_POP]	POP switch homing
	D3DONE[MAX_AT] [ATPOPMAX]	a logical control variable to avoid double counting AT to POP trunk groups
	TOT_IMT	total number of backbone trunks at the switch
	EOLOC[MAX_EO] [2]	location of each EO
	ATLOC[MAX_AT] [2]	location of each AT
	SWLOC[MAX_SWITCH] [2]	location of each SW
	POPLOC[MAX_POP] [2]	location of each POP
real	IMTTraf[MAX_SWITCH] [MAX_SWITCH]	traffic between switches

	TEP[MAX_EO]	traffic EO to POP
	TEA[MAX_EO]	traffic EO to AT
	TAP[MAX_AT][ATPOPMAX]	traffic EO to POP
	ACSlec[MAX_SWITCH]	LEC access capacity at the switch. This is the sum of the D3[POP] s for POPs homed to SW. This is a function of TEP and TAP. Alternatively, ACSlec is the sum of AC[k] for all EOs homed to SW. This is computed before damage.
	ACSiec[MAX_SWITCH]	IEC access capacity at the switch. ACSiec is used to compute ACSPOP. ACSiec is a function of IMTTraf.
		AC[MAX_EO] total access capacity
	D3[MAX_POP]	access traffic engineered from LEC switches to POP. This is the access traffic arriving at POP from the EO-POP and AT-POP TGs homed to it. D3 is a function of TEP and TAP. This is equal to intraswitch plus interswitch traffic at a POP
	ACPiec[MAX_POP]	access traffic engineered from IEC switches to POP
	PropIntra[MAX_SWITCH]	proportion of intraswitch traffic
	fProp[MAX_SWITCH]	ratio intraswitch traffic to interswitch traffic
	Overflow[MAX_EO]	traffic overflowing AT to POP on high usage trunks
character	swclli[MAX_SWITCH][12]	matrix of clli codes for each switch

global variables defined in "keyfiles.h"

FILE	*fileptr	points to an open input file
	*outfile	points to an open output file
character	eo_file[80]	name of the EO file
	at_file[80]	name of the AT file
	iec_eo_file[80]	name of the IEC EO file
	iec_at_file[80]	name of the IEC AT file
	iec_sw_file[80]	name of the IEC SW file
	iec_pop_file[80]	name of the IEC POP file

Local Variables

Variables local to main()

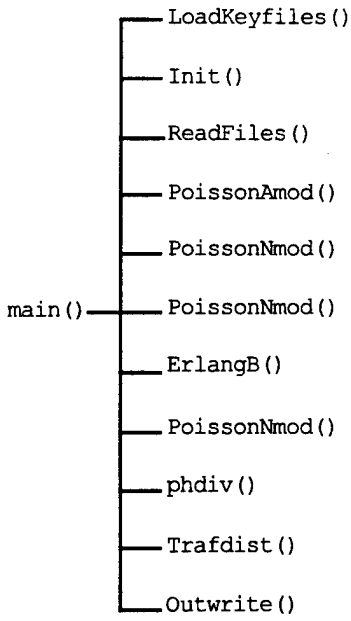
extern	character	*optarg	a string containing a single command line argument.
	integer	optind	the number of single command line arguments to be processed, supplied externally by the operating system
character	c		a command line option character
	outfile[80]		the name of the output file
	line[200]		buffer for parsing a single input line from a file
	livekeyfile[80]		the name of the input key file for the live networks
	**argv[ ]		an array containing all of the command line arguments
integer	err=0		error flag indicating a problem with the command line arguments
	otog=0		indicates use of the [-o] option (a user supplied output file name)

	`ttog=0`	command line parameter setting indicates the use of the [-t] option (currently not used)
	`htog=0`	indicates the [-h] option (use of new high usage rules, this is now the default setting)
	`SWA`	the index of a switch for printing of debug statements in `TrafDist()`
	`test`	not used
	`AC_tot=0`	sums the total access capacity across all EOs
	`t3_num=0`	the number of end offices that overflow from HU to final homing
	`ovflw_count=0`	the number of end offices that overflow from HU to final homing (same as `t3_num`)
	`Dummy`	used to read in a dummy integer value
	`i,j,k,l,m,n,pm`	loop count variables
	`SEP_total=0`	sums the number of high usage trunks between EOs and POPs
	`argc`	the number of command line arguments
real	`ovflw_total=0`	the total amount of traffic which overflows from the high usage trunks
	`TEP_total=0`	the total amount of traffic between EOs and POPs
	`TEA_total=0`	the total amount of traffic between EOs and ATs
	`wovflw_total=0`	the weighted overflow based on access capacity across all EOs
	`wovflw_denom=0`	the total access capacity across all EOs

Component Functions

void	`Init()`	initializes all network variables
	`ReadFiles()`	reads in each section of the input files specified in the `<keyfile>`
	`Trafdist()`	not part of the `attlive` algorithm (test function)
	`OutWrite()`	writes out each section of the output file
	defined in "`attlive.h`"	
void	`LoadKeyfile()`	loads input file names from the `<keyfile>` specified in the command line
double	`ErlangB()`	traffic calculator for engineering overflow capacity using ErlangB blocking
	`PoissonA()`	traffic calculator for engineering traffic from switch A to switch B and vice-versa
	`PoissonAmod()`	traffic calculator for engineering traffic from switch A to switch B and vice-versa
	`PoissonNmod()`	traffic calculator for engineering number of trunks
integer	`PoissonN(Ain,Bin)`	traffic calculator for engineering number of trunks
	`phdiv(eo,at,pop)`	determines physical diversity based on collocation

Function Tree



Algorithmic Description

This module completes the representation of the AT&T network and the traffic that is carried on that network. The procedure is to distribute the AT&T traffic down though the LECs then reverse engineer the sizes of the TGs needed to carry this traffic in order to build a reasonable model of the network and its relationship to the LECs. A similar process will be undertaken with MCI and Sprint.

The `main()` routine contains most of the algorithmic code for this module. After the command line toggles have been interpreted, the `main()` routine begins by passing the `<keyfile>` argument into `LoadKeyfile()`, which reads in the names of the LEC and IEC (AT&T) files. It then calls `Init()`, which initializes all network variables. `Readfile()` is called next to read in all network data. `Main()` then opens the `<outfile>` and proceeds to perform the core of the algorithmic work in this module, in the following order:

1) Call `PoissonAmod()` which uses the number of backbone/IEC trunks connecting switch A to switch B (and B to A) to compute the traffic across the trunks from A to B (and B to A).

2) Count the total number of central office codes associated with the end office homed to each switch.

3) Engineer total access traffic (two-way) between the LEC and the switch and engineer total access traffic (two-way) between the switch and the IMTs by summing traffic from all backbone trunks that end at a switch. (These two values are equal if `fprop` is disabled, i.e. set equal to 1.)
Keep a running total of the number of IMTs and total traffic carried by them.

4) Compute the average access capacity of each end office by multiplying the total traffic at each switch by the ratio: *number of EO COCs/total number of COCs;*

Keep a running total of the overall access capacity.

5) Engineer three types of LEC trunk group size homings and the traffic carried on them;
Sum EO-AT traffic up to AT-POP traffic; and
Keep diagnostics.

Type 1. HU-only homing.
Engineer the size of the trunk groups that send traffic from the end office to the switch.
Call `PoissonNmod()` to engineer the size of the trunk groups.
Set `traffic = access capacity`

Type 2. AT-only homing.
Engineer the traffic sent from the end office to the Access Tandem and then to the switch.
Calculate EO-AT traffic.
Sum EO-AT traffic up to AT-POP traffic.

Type 3. HU with AT overflow.
Engineer the size of the trunk groups that use a combination of **Type 1** and **Type 2**
Call `PoissonNmod()` to engineer the number of trunks needed to send communications directly to the switch.
Call `ErlangB()` to engineer overflow capacity sent first through the AT and then to the switch.
Calculate traffic EO-AT as `AC* (Overflow traffic)`.
Calculate traffic EO-POP as `AC* (1-Overflow traffic)`.
Sum traffic EO-AT up to traffic AT-POP.

6) Reduce access capacity by 1/2 to capture access capacity only, assuming access = egress at an EO. Perform this for each EO in the LEC and for each switch in the IEC.

7) Use AT-POP Traffic and `PoissonNmod()` to calculate the number of AT-POP trunks.

8) For each Type 3 homing (HU with AT overflow), call `phdiv()` determine physical diversity. (i.e. collocated = not physically diverse; not collocated = physically diverse). This is necessary to determine if physical damage to an AT site will also cause damage to the HU homing.

9) Calculate AT traffic engineered between LEC switches and POP.
Sum HU traffic up to POPs.
Sum AT traffic up to POPs

10) Calculate the amount of interswitch access traffic at a POP.
Determine intraswitch traffic.

11) Calculate the proportion of intraswitch traffic

12) Calculate one-half the value of: traffic EO to POP and traffic EO to AT.

At this point the calculation of Traffic and the determination of Trunks is complete in the LEC

13) Check for EOs with `AC = 0`. Print warning if found.

14) If `ttog` set 'on' then call `TrafDist()`. This optional routine is not part of TAMI but was used to perform testing of best modeling assumptions. This routine attempts to define the community of interest weighting that describes how much or how little traffic a switch will send to another switch due to their proximity.

15) Call `OutWrite()` which formats and writes all variables to the outfile which is a LECAM input file.

Inputs none; operates on global variables

Outputs

global	integer	NUMEO=0	number of end offices
		NUMAT=0	number of access tandems
		NUMPOP=0	number of points of presence
		NUMSW=0	number of switches
		TOT_IMT=0	total number of backbone trunks at the switch
		AT[i] =0	traffic at the AT
		ATPOP[i] =0	traffic between the AT and the POP
		HUPOP[i] =0	traffic between the high usage AT and the POP
		TYPE[i] =0	type of homing between the AT and the SW
		SW[i] =0	traffic at the switch
		LATA[i] =0	LATA each EO is homed to
		COC[i] =0	central office code homings
		SEP[i] =0	size of trunks between EO and POP
		PhysDiv[i] =0	physical diversity value for each EO of TYPE = 3
		F0[i][0] =0	engineered traffic offered at at an EO
		F0[i][1] =0	engineered traffic offered at at an EO
		Overflow[i] =0	traffic overflowing AT to POP on high usage trunks
		D3DONE[i][j] =0	a logical control variable to avoid double counting AT to POP trunk groups
			HAP[i][j] =0 number of high usage trunks between the AT and the POP
		SAP[i][j] =0	size of trunks between AT and POP
		fProp[i] =0	ratio intraswitch traffic to interswitch traffic
		SW_nlata[i] =0	number of LATAs that a switch serves
		COCSW[i] =0	matrix of homings betwen switches and COCs
		ACSlec[i] =0	LEC access capacity at the switch
		ACSiec[i] =0	IEC access capacity at the switch
		PropIntra[i] =0	proportion of intraswitch traffic
		IMT[i][j] =0	matrix of backbone trunk traffic
global	real	TEP[i] =0.0	traffic EO to POP
		TEA[i] =0.0	traffic EO to AT
		AC[i] =0.0	total access capacity
		TAP[i][j] =0.0	traffic AT to POP
		D3[i] =0.0	access traffic engineered from LEC switches to POP
		ACPiec[i] =0.0	access traffic engineered from IEC switches to POP
		IMTTraf[i][j] =0.0	traffic between switches

returns no formal returns

Purpose Initializes all network variables

Called By `main()`

Calls To none

Local Variables

integer	i, j	loop count variables

Global Variables none

Global Constants

MAX_EO	maximum number of end offices
MAX_AT	maximum number of access tandems
ATPOPSIZE	the number of trunks between the AT and the POP
MAX_POP	maximum number of points of presence
MAX_SWITCH	maximum number of switches

Algorithmic Description For every EO this function sets the trunks sizes and traffic carried between EOs, ATs, POPs, SWs to initial values (0) and initializes other settings for ATs, POPs, SWs and traffic in the backbone. Initializes all network variables.

Inputs

file	iec_eo_file	IEC EO file
	iec_at_file	IEC AT file
	iec_pop_file	IEC POP file
	qlink_file	QTCM link file
	eo_file	EO file
	at_file	AT file

Outputs

character	AT[ ]	traffic at the AT
	ATPOP[ ]	traffic between the AT and the POP
	HUPOP[ ]	traffic between the high usage AT and the POP
	TYPE[ ]	type of homing between the AT and the SW
	SW[ ]	traffic at the switch
	LATA[ ]	traffic at the LATA
	COC[ ]	central office code homings
	clli	holds a clli code
	equip	holds an equipment code
	swclli[ ][ ]	matrix of clli codes for each switch
integer	HAP[ ][ ]	number of high usage trunks between the AT and the POP
	EOLOC[ ][ ]	location of each EO
	ATLOC[ ][ ]	location of each AT
	SWLOC[ ][ ]	location of each switch
	SW_nlata[ ]	number of LATAs that a switch serves
	POPLOC[ ]	location of each POP
	popswitch[ ]	homing for each POP switch
	IMT[ ][ ]	matrix of backbone trunk traffic

Purpose to read in all live network variables

Called By main()

Calls To none

Local Variables

character	line[200]	
integer	cap	trunk size capacity
	dummy	dummy variable
	pop_length	POP length
	cap_iec_tot	total IEC capacity
	i,j,k,l,m,n,p	loop variables

Global Variables

integer	NUMEO	number of end offices

NUMAT	number of access tandems
NUMSW	number of switches
ATPOPSIZE	the number of trunks between the AT and the POP

Algorithmic Description

This function reads in the live network variables from several input files. The input is performed in the following sequence:

1) Read in LEC data from IEC EO file which contains homing chains from EO-SW and count number of EOs

2) Read in LEC data from the IEC AT (HAP) file which describes AT-POP homings, which may be multiple.

3) Read in EO V-H coordinates from the EO file.

4) Read in AT V-H coordinates from the AT file.

5) Read in Switch V-H coordinates from the IEC switch file. Also read in and store the CLLI code and the number of LATAs that a switch serves

6) Read in POP V-H coordinates from the IEC POP file. Also read in POP switch homing.

7) Read the undamaged IMT matrix from qlink. This is the undamaged matrix of IMT capacity. Divide each one-way TG into 50% i->j and 50% j->i to account for bi-directional traffic.

Inputs

integer	SWA	the index of a switch for printing of debug statements

Outputs

returns none

Purpose This optional routine is not part of TAMI but was used to perform testing of best modeling assumptions.

Called By main()

Calls To none

Local Variables

integer	i,j	loop count variable
real	vdiff	vertical difference
	hdiff	horizontal difference
	dist	distance between switches of interest
	weight	a weighting factor
	Sum[ ]	sum of access traffic of all switches

Global Variables

character	swclli[ ][ ]	matrix of clli codes for each switch
integer	NUMSW=0	number of switches
real	IMTTraf[ ][ ]	traffic between switches
	ACSiec[ ]	IEC access capacity at the switch
	SWLOC[ ][ ]	location of each SW

Algorithmic Description

This routine defines the community of interest weighting that describes how much or little traffic a switch will send another due to their proximity. The approach is to assume that equidistant switches would send each other traffic proportional to the ACSiec of both the source and destination switches:

t[i][j] = ACSiec[i] * ACSiec[j]/(SUM(k=1..NUMSW; k!=i) ACSiec[k])

This quantity ignores community of interest, so it can be modified as follows:

t[i][j] = ACSiec[i] * (ACSiec[j] * func(distance)) /
(SUM(k=1..NUMSW; k!=i) {ACSiec[k] * func(distance)})

The func(distance) can be any type of weighting function. Here it is assumed that it will be some function inversely proportional to the distance between the two switches. For the purposes of this routine, func(distance) is estimated as follows:

t[i][j] = ACSiec[i] * (ACSiec[j] * func(distance)) /
(SUM(k=1..NUMSW; k!=i) {ACSiec[k]})

where the sum in the denominator is not weighted. Therefore, the weight is approximated by:

func(distance) = (SUM(k-1..NUMSW; k!=i) {ACSiec[k]} * t[i][j] /ACSiec[i] * ACSiec[j]

Note: `IMTTraf[ i][ j]` is equal to t[i][j] in the code.

After declaring local variables and initializing values, the function proceeds to execute the following steps:

1) Compute sum of the access traffic for all unique switches

2) Compute results for all switch pairs

3) Calculate distance from switch A to all others
Calculate estimate of weight
Print selected results

4) Compute results for switch pairs from switch A to all other switches, printing all the results

5) Calculate distance from switch A to all others
Calculate estimate of weight
Print all results.

Inputs none

Outputs please see description of output file "`att2lecam file`" under the module description section.

Purpose to format and print data for input to lecam.c

Called By `main()`

Calls To none

Local Variables

integer	`count, k, l, m, n`	loop count variables

Global Variables

integer	NUMEO	number of end offices
	NUMAT	number of access tandems
	NUMPOP	number of points of presence
	NUMSW	number of switches

Algorithmic Description This function writes out all output variables to the output file "`att2lecam.step1`" for a list of these variables and the file format see the description of the output file under the module description section.

3.2 mcilive: Build MCI Live Network and Traffic

Purpose

The purpose of this module is to complete the representation of the undamaged MCI network. This module uses existing publicly available data on that network along with telco engineering and reverse engineering methods to completely represent the MCI network from the TGs down to the LECS.

This program also allocates traffic from the switches down to the EOs, creating traffic patterns along the way on EO-POP, AT-POP, and EO-AT TGs and builds a traffic matrix. This module is used in conjunction with `mciwdmg` which introduces damage effects into the network data files. The final output of the two step process after the `mciwdmg` module is input into the LECAM module.

Call Syntax

`mcilive -k <key file> [ mandatories][ options]`

mandatory:	*special syntax:*	*function:*
`-k`	`<keyfile>`	reads in input file `<key file>`

optional:	*special syntax:*	*function:*
`-o`	`<outfile>`	reads in output file name. If not specified, default file name is `mci2lecam.step1`
`-t`	`<#>`	invokes `Trafdist()`, traffic distribution matrix routine, default setting
`-h`		invokes new High Usage occupancy and overflow algorithm commands (this setting is now considered the default setting
`-p`		High Usage overflow default setting override
`-?`		user help—prints call syntax and exits w. without running

example `mcilive -k dplive.key -h` (spaces optional)

Input Files

key file — This file contains hardcoded instructions for opening up and reading in the various data files to be used in this module.

format — **SEE SECTION 2.1/EXHIBIT 2-3**

Output Files

mci2lecam file — This file contains the formatted output data to be used in the LECAM module

format
header:

* MCI live data file (damage added in MC loop)
<IEC label>
("IECid=", c3)
<switch size>
("SWITCH_SIZE", i3)
<POP size>
("POP_SIZE", i3)
<AT size>
("AT_SIZE", i4)
<EO size>
("EO_SIZE", i5)
<AT to POP size>
("ATPOPSIZE", i3)

EOAT section 1:	<traffic at the AT for each EO>, (i4, I4,.....) in rows of 20
HUPOP section 2:	<traffic between high usage AT and the POP for each EO>, (i4, I4,.....) in rows of 20
ATPOP section 3:	<traffic between AT and the POP for each EO>, (i4, I4,.....) in rows of 20
HAP section 4:	<number of high usage trunks AT and the POP for each EO>, (i4, I4,.....) in rows of 20
SDlive section 5:	< size of trunks between EO and POP>, (i5, I5,.....) in rows of 15
TEP section 6:	<traffic between EO and the POP for each EO>, (f8.2, f8.2, ...) in rows of 10
TAP section 7:	<traffic between AT and the POP for each EO>, (f8.2, f8.2, ...) in rows of 10
F0 section 8:	<number of high usage trunks AT and the POP for each EO>, (f14.8, f14.8, ...) in rows of 2
AClive section 9:	<total access capacity>, (f.8, f.8, ...) in rows of 5
ACSlive section 10:	< total access capacity for the LEC>, (f.8, f.8, ...) in rows of 5
ELAP section 11:	<traffic between AT and the POP for each AT>, (f8.2, f8.2, ...) in rows of 6
SAPlive section 12:	<size of trunks between AT and the POP for each AT>, (i5, I5,.....) in rows of 6
D3 section 13:	< access traffic engineered from LEC switches to POP>, (f8.2, f8.2, ...) in rows of 10
ACSPOP section 14:	< access traffic engineered from IEC switches to POP>, (f8.2, f8.2, ...) in rows of 10
POPSWITCH section 15:	< POP switch homing>, (i4, i4, ...) in rows of 20
EOlata section 16:	< LATA each EO is homed to >, (i4, i4, i4, ...) in rows of 20
PhysDiv section 17:	<physical diversity value for each end office: 0/1>, (i1, i1, ...) in rows of 80
IMT section 18:	< matrix of backbone trunk traffic for each switch>, (i5, i5, ...) in rows of 15
PropIntra section 19:	< proportion of intra-switch traffic for each switch>, (f.8, f.8, ...) in rows of 5

Includes

"mcilive.h"	Defines global network and traffic variables and constants
"lecam.h"	Defines network sizing constants
"fileio.c"	User-defined I/O functions; see Appendix A

Constants

Constants used in mcilive and defined in "mcilive.h":

OVFLW	0.25	MCI overflows 75% from HU to final
TRKOCC	0.95	HUs are 95% occupied
NODIRECT_WEIGHT	0.3	indicates no direct weight value
MAX_ITER	100	maximum number of iterations for traffic algorithm
TOLERANCE	0.1	precision level

Constants used in mcilive and defined in "lecam.h":

TRUE	1	true
FALSE	0	false
MAX_AT	1000	maximum number of access tandems
MAX_EO	19250	maximum number of end offices
ATPOPMAX	9	maximum number of POPs a single AT may home to
MAX_POP	575	maximum number of points of presence
MAX_SWITCH	130	maximum number of switches
MAX_LATA	1000	maximum number of LATAs
MAX_REGION	5	maximum number of regions covered by a LATA

Global Variables

global variables defined in "mcilive.h"

integer

KEYTOG	toggle to indicate input 'file name' was read in successfully
GSEP	given size of EO to POP connection
GSAP	given size of AT to POP connection
NUMEO	number of end offices
NUMAT	number of access tandems
NUMPOP	number of points of presence
NUMSW	number of switches
ATPOPSIZE	the number of trunks between the AT and the POP
HAP[MAX_AT] [ATPOPMAX]	number of high usage trunks between the AT and the POP
AT[MAX_EO]	traffic at the AT
ATPOP[MAX_EO]	traffic between the AT and the POP
HUPOP[MAX_EO]	traffic between the high usage AT and the POP
TYPE[MAX_EO]	type of homing between the AT and the SW
SW[MAX_EO]	traffic at the switch
LATA[MAX_EO]	traffic at the LATA
COC[MAX_EO]	central office code homings
IMT[MAX_SWITCH] [MAX_SWITCH]	matrix of backbone trunk traffic
SEP[MAX_EO]	size of trunks between EO and POP
SAP[MAX_AT] [ATPOPMAX]	size of trunks between AT and POP
COCSW[MAX_SWITCH]	matrix of homings between switches and COCs
PhysDiv[MAX_EO]	physical diversity value for each EO of TYPE = 3
SW_nlata[MAX_SWITCH]	number of LATAs that a switch serves
popswitch[MAX_POP]	POP switch homing
D3DONE[MAX_AT] [ATPOPMAX]	a logical control variable to avoid double counting AT to POP trunks groups

	TOT_IMT	total number of backbone trunks at the switch
	EOLOC[MAX_EO] [2]	location of each EO
	ATLOC[MAX_AT] [2]	location of each AT
	SWLOC[MAX_SWITCH] [2]	location of each SW
	POPLOC[MAX_POP] [2]	location of each POP
	TAP_flag[MAX_AT] [ATPOPMAX]	traffic indicator, AT to POP
real	IMTTraf[MAX_SWITCH] [MAX_SWITCH]	traffic between switches
	TEP[MAX_EO]	traffic EO to POP
	TEA[MAX_EO]	traffic EO to AT
	TAP[MAX_AT] [ATPOPMAX]	traffic EO to POP
	ACSlec[MAX_SWITCH]	LEC access capacity at the switch ACS[SW] derived by summing the P.01 engineered traffic on IMTs going into SW, times 0.5 for one-way. This parameter is later decremented in the allocation algorithm, and is thus unsuitable for the calculations that follow that algorithm. ACS is a function of IMTTraf.
	ACSiec[MAX_SWITCH]	exact copy of ACS[SW], but is not decremented in the allocation algorithm. ACSiec is used to compute ACSPOP. ACSiec is a function of IMTTraf.
	AC[MAX_EO]	total access capacity
	D3[MAX_POP]	access traffic arriving at POP from the EO-POP and AT-POP TGs homed to it. D3 is a function of TEP and TAP. This is equal to intraswitch plus interswitch traffic at a POP
	ACSPOP[POP]	access traffic that POP can offer to SW for transmission across IMTs. This is D3 * (ACSiec/ACSlec), and is written to outfile as the "real" D3, since it more accurately reflects engineered load at a pop. ACSPOP is a function of IMTTraf. It is computed before D3 is decremented for damage, so that it reflects true all-alive engineered traffic. Is equal to inter switch traffic (only) at a POP.
	ACSlec[SW]	sum of the D3[POP] s for POPs homed to SW. This is a function of TEP and TAP. Alternatively, ACSlec is the sum of AC[k] for all EOs k homed to SW. This is computed before damage.
	ACPiec[MAX_POP]	access traffic engineered from IEC switches to POP
	PropIntra[MAX_SWITCH]	proportion of intraswitch traffic
	fProp[MAX_SWITCH]	ratio intraswitch traffic to interswitch traffic
	Overflow[MAX_EO]	traffic overflowing AT to POP on high usage trunks
	TAPA[MAX_AT] [ATPOPMAX]	traffic AT to POP
	TAP_target[MAX_AT] [ATPOPMAX]	target amount of traffic AT to POP
double	FO[MAX_EO] [2]	divided into HU traffic (0) and FINAL traffic (1)
character	swclli[MAX_SWITCH] [12]	matrix of clli codes for each switch
	clli[12]	individual clli code
	equip[4]	four character equipment code

global variables defined in "keyfiles.h"

FILE	`*fileptr`	points to an open input file
	`*outfile`	points to an open output file
	`*fptr`	points to an open input file in `TrafDist()`
character	`eo_file[ 80]`	name of the EO file
	`at_file[ 80]`	name of the AT file
	`iec_eo_file[ 80]`	name of the IEC EO file
	`iec_at_file[ 80]`	name of the IEC AT file
	`iec_sw_file[ 80]`	name of the IEC SW file
	`iec_pop_file[ 80]`	name of the IEC POP file
	`morph_file[ 80]`	name of the morphing file
	`swhmg_file[ 80]`	name of the switch homing file

Local Variables Variables local to `main()`

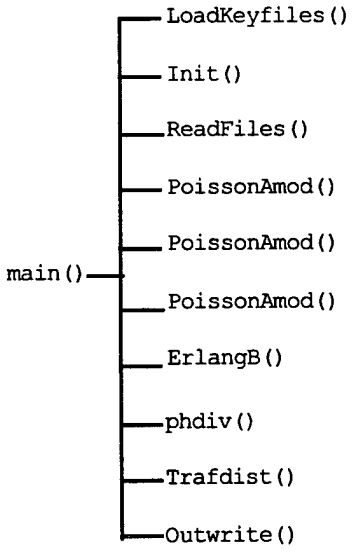
extern	character `*optarg`	a string containing a single command line argument.
	integer `optind`	the number of single command line arguments to be processed, supplied externally by the operating system
character	`c`	a command line option character
	`outfile[ 80]`	the name of the output file
	`line[ 200]`	buffer for parsing a single input line from a file
	`livekeyfile[ 80]`	the name of the input key file for the live networks
	`**argv[ ]`	an array containing all of the command line arguments
integer	`err=0`	error flag indicating a problem with the command line arguments
	`otog=0`	indicates use of the [-o] option (a user supplied output file name)
	`ttog=0`	command line parameter setting indicates the use of the [-t] option (currently not used)
	`htog=0`	indicates the [-h] option (use of new high usage rules, this is now the default setting)
	`ptog=0`	indicates the [-p] option (override use of new high usage overflow rules)
	`test`	not used
	`AC_tot=0`	sums the total access capacity across all EOs
	`t3_num=0`	the number of end offices that overflow from HU to final homing
	`ovflw_count`	the number of end offices that overflow from HU to final homing (same as `t3_num`)
	`dummy`	used to read in a dummy integer value
	`i,j,k,l,m,n,p,pm`	loop count variables
	`SEP_total=0`	sums the number of high usage trunks between EOs and POPs`count`count variable
	`argc`	the number of command line arguments
structure	`AT_hmg[ MAX_AT]`	of type `AT_hmg_struct` *with fields:*
integer	`type2`	number of Type 2 homing switches
	`type3`	number of Type 3 homing switches
real	`ovflw_total`	the total amount of traffic which overflows from the high usage trunks
	`TEP_total`	the total amount of traffic between EOs and POPs
	`TEA_total`	the total amount of traffic between EOs and ATs

	wovflw_total	the weighted overflow based on access capacity across all EOs
	wovflw_denom	the total access capacity across all EOs
	sum2_AC=0	sum of the access capacity for Type 2 switches
	sum3_AC=0	sum of the access capacity for Type 3 switches

Component Functions

void	Init()	initializes all network variables
	ReadFiles()	reads in each section of the input files specified in the <keyfile>
	Trafdist()	This routine defines the community of interest weighting that describes how much or how little traffic a switch will send to another switch due to their proximity.
	OutWrite()	writes out each section of the output file defined in "mcilive.h"
void	LoadKeyfile()	loads input file names from the <keyfile> specified in the command line
double	ErlangB()	traffic calculator for engineering overflow capacity using ErlangB blocking
	PoissonA()	traffic calculator for engineering traffic from switchA to switchB and vice-versa
	PoissonAmod()	traffic calculator for engineering traffic from switchA to switchB and vice-versa
integer	phdiv()	determines physical diversity based on collocation

Function Tree



Algorithmic Description

This module completes the representation of the MCI network and the traffic that is carried on that network. The procedure is to distribute the MCI traffic down though the local exchange carriers (LEC) in order to build a reasonable model of the network and its relationship to the LECs. A similar process is undertaken with AT&T and Sprint.

MCI provides the sizes of High Usage trunks (HU) and access tandems (AT) to point-of-presence (POP) trunk groups (TG), which are used to engineer traffic in the LECs. Where homings through an AT are involved, traffic must be allocated down from the AT to the end offices (EO). There are several ways to do this, each with drawbacks and advantages. This module uses the most accurate method where possible; cases where this method breaks down are detected, and a secondary method is used to allocate traffic from these ATs down to their EOs.

The `main()` routine contains the majority of the algorithmic code for this module. After the command line toggles have been interpreted, the `main()` routine begins by passing the `<keyfile>` argument into `LoadKeyfile()`, which reads in the names of the LEC and interexchange (IEC) files. It then calls `Init()`, which initializes all network variables. `Readfile()` is called next to read in all network data. `Main()` then opens the `<outfile>` and proceeds to perform the core of the algorithmic work in this module, in the following order:

1) Call `PoissonAmod()` which uses the number of backbone/IEC trunks to compute the backbone traffic across the trunks.

2) Count the total number of central office codes associated with the end office homed to each switch.

3) Engineer total interswitch traffic at each switch by summing traffic from all backbone trunks that end at a switch. (Since MCI trunks are not bi-direcitonal, must add access and egress backbone traffic at a switch.)
   Set `access traffic` = to one half of total traffic.

4) Use a **3-Pass** algorithm to allocate traffic to individual LEC network segments, for 3 types of LEC trunk group size homings: **Type 1**, HU-only homings, **Type 2**, AT-only homings, and **Type 3**, homings that involve an HU overflowing to the AT (final) homing :

In general calculate:

- AC (access traffic offered by an EO)
- TEA (traffic going on the EO-AT TG, if any)
- TAP (traffic going on the AT-POP TG, if any)
- TEP (traffic going on the EO-POP TG, if any).

PASS 1 Engineer the straightforward cases and set up statistics to support the more complex cases.

Type 1. HU-only homing.
Call `PoissonAmod()` to engineer AC directly from the EO-POP TG size, assuming P.01, (1% blocking).
Set EO-POP traffic equal to AC.
De-allocate EO AC from switch (SW) traffic.

Type 2. AT-only homing.
Traffic over AT-only homings cannot be allocated until the effect of overflow from HU trunks is determined. In the first pass we sum up central office codes (COC) for all EOs homed to the same AT-POP TG.

Type 3. HU with AT overflow.
Traffic for HU homings that overflow through the AT is calculated using the methodology shown below. In later passes, it is determined if this method broke down for a particular AT. If so, an alternative algorithm is used. The preferred algorithm is:

A) Call `PoissonAmod()` to engineer communications sent directly EO-POP (with P.80 blocking on the HU TG meaning the last trunk in the TG is occupied 80% of the time).

B) Call `ErlangB()` to engineer overflow capacity sent first through the AT then to the SW. Traffic overflowing to the AT is computed by offering AC to the HU TG using Erlang B blocking. This Erlang B blockage is used to determine traffic carried on the HU, and traffic that must overflow from the EO to the AT.

C) Compute the running sum of EO-AT traffic for each AT-POP TG to determine if our methodology would create more AT-POP traffic than could be handled by the AT-POP TG. As a precautionary measure, compute the variables required by the backup algorithm: the sum of COCs and overflow-weighted COCs to each AT-POP TG for the **Type 3** homings.

PASS 1.5 This pass only applies to **Type 3** EOs,
No network variables are calculated. Only determines if first-choice numbers in previous pass are valid.

Type 1. HU-only homing. No calculations for this pass.

Type 2. AT-only homing. No calculations for this pass.

Type 3. HU with AT overflow.
Determines whether the first or second-choice algorithm will be used for allocating AT-POP traffic down to EOs. If the first-choice algorithm won't work (i.e. causes AT-POP traffic to be greater than what the AT-POP TG can accommodate), then a flag is set for the next pass, and the **Type 3** overflow-weighted COCs are combined with the **Type 2** COCs to create the total needed by the alternative algorithm.
If the first choice algorithm will work, then the share of AT-POP traffic already allocated to **Type 3** EOs is subtracted from the amount available to be allocated to the **Type 2** EOs that use the same AT.

PASS 2 The second pass allocates traffic to the **Type 2** EOs (AT-only homings) regardless of the method used for **Type 3** traffic allocation. Calculate traffic the **Type 3** EOs for the cases where the alternative allocation algorithm must be used.

Type 1. HU-only homing. No calculations for this pass.

Type 2. AT-only homing.
Use COCs and the AT-POP traffic to engineer the access capacity and the EO-AT traffic and then to the SW.
Calculate AC based on COCs
Allocate traffic AT-POP
Calculate traffic EO-AT
De-allocate AC traffic at the SW

Type 3. HU with AT overflow.
Use second-choice algorithm:

If htog is set to true
Allocate AT-POP traffic based on overflow weighted COCs
Replace previous values of TEA and AC
De-allocate AC traffic at the SW

else
Allocate AT-POP traffic based on HU overflow weighted COCs

De-allocate AC traffic at the SW
Replace previous values of EO-AT traffic and AC.

At this point all LEC network traffic variables have been engineered and the calculation of Traffic and the determination of Trunks is complete in the LEC .

5) Reduce AC by 1/2 to capture access capacity only, assuming access = egress at an EO. Perform this for each EO in the LEC and for each SW in the IEC.
Sum traffic from LECs up to SW (`ACSlec`).

6) For each **Type 3** homing (HU with AT overflow), call `phdiv()` to determine physical diversity. (i.e. collocated = not physically diverse; not collocated = physically diverse). This is necessary to determine if physical damage to an AT site will also cause damage to the HU homing.

7) Compute and print overflow and occupancy summary statistics:
Sum AC for each **Type 2.**
Sum **Type 2** homings.
Verify actual engineered overflow form the HU TG to the POP.
Sum AC for each **Type 3.**
Calculate average and weighted average overflow.
Calculate EO-POP and EO-AT traffic.

8) Use `ACSlec` and results of `phdiv()` to recalculate SW AC. Backbone trunks that connect to a SW with no EO homings should not be included in calculating SW AC. This occurs when a SW is collocated.

9) Calculate AT traffic engineered between LEC switches and POP.
Sum HU traffic up to POPs.
Sum AT traffic up to POPs

Switch and POP variables are now calculated, including versions that include interswitch traffic only, and versions that include both inter and intraswitch traffic. From this, the proportion of intraswitch traffic at each switch (`PropIntra`) is calculated.

10) Calculate the amount of interswitch access traffic at a POP.
Determine intraswitch traffic.

11) Calculate the proportion of intraswitch traffic

12) Calculate one-half the value of : traffic EO to POP and traffic EO to AT.

At this point the calculation of Traffic and the determination of Trunks is completed in the LEC

13) If `ttog` set true then call `TrafDist()`.This routine defineS the community of interest weighting that describes how much or how little traffic a switch will send to another switch due to their proximity.

14) Call `OutWrite()` which formats and writes all variable to the outfile which is a LECAM input file.

Inputs none; operates on global variables

Outputs

global	integer	NUMEO=0	number of end offices
		NUMAT=0	number of access tandems
		NUMPOP=0	number of points of presence
		NUMSW=0	number of switches
		TOT_IMT=0	total number of backbone trunks at the switch
		GSEP=0	given size of EO to POP connection
		GSAP=0	given size of AT to POP connection
		TAP_target=0	target amount of traffic AT TO POP
		TAP_flag=0	indicates target traffic has been reache for AT to POP
		COC_type3=0	number of Type 3 COCs
		COCEFF_type3=0	effective number of Type 3 COCs
		AT[i] =0	traffic at the AT
		ATPOP[i] =0	traffic between the AT and the POP
		HUPOP[i] =0	traffic between the high usage AT and the POP
		TYPE[i] =0	type of homing between the AT and the SW
		SW[i] =0	traffic at the switch
		LATA[i] =0	LATA each EO is homed to
		COC[i] =0	central office code homings
		SEP[i] =0	size of trunks between EO and POP
		PhysDiv[i] =0	physical diversity value for each EO of TYPE = 3
		F0[i][0] =0	engineered traffic offered at an EO
		F0[i][1] =0	engineered traffic offered at an EO
		Overflow[i] =0	traffic overflowing AT to POP on high usage trunks
		D3DONE[i][j] =0	a logical control variable to avoid double counting AT to POP trunk groups
		HAP[i][j] =0	number of high usage trunks between the AT and the POP
		SAP[i][j] =0	size of trunks between AT and POP
		COCSW[i] =0	matrix of homings between switches and COCs
		ACSlec[i] =0	LEC access capacity at the switch
		ACSiec[i] =0	IEC access capacity at the switch
		PropIntra[i] =0	proportion of intra-switch traffic
		IMT[i][j] =0	matrix of backbone trunk traffic
global	real	TEP[i] =0.0	traffic EO to POP
		TEA[i] =0.0	traffic EO to AT
		AC[I] =0.0	total access capacity
		TAP[i][j] =0.0	traffic AT to POP
		D3[i] =0.0	access traffic engineered from LEC switches to POP
		COCEFF=0.0	effective number of COCs
		ACSPOP=0.0	access capacity AT to POP
		TAPA=0.0	traffic AT to POP
		ACS=0.0	access capacity at the switch popswitch= (-1) initial POP to SW setting
		IMTTraf[i][j] =0.0	traffic between switches

returns no formal returns

Purpose Initializes all network variables

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	loop count variables

Global Variables none

Global Constants

`MAX_EO`	maximum number of end offices
`MAX_AT`	maximum number of access tandems
`ATPOPSIZE`	the number of trunks between the AT and the POP
`MAX_POP`	maximum number of points of presence
`MAX_SWITCH`	maximum number of switches

Algorithmic Description For every EO this function sets the trunk sizes and traffic carried between EOs, ATs, POPs, SWs to initial values (0) and does other things for ATs, POPs, SWs and traffic in backbone. Initializes all network variables.

Inputs

file	iec_eo_file	IEC EO file
	iec_at_file	IEC AT file
	iec_pop_file	IEC POP file
	qlink_file	QTCM link file
	eo_file	EO file
	at_file	AT file

Outputs

character	AT[ ]	traffic at the AT
	ATPOP[ ]	traffic between the AT and the POP
	HUPOP[ ]	traffic between the high usage AT and the POP
	TYPE[ ]	type of homing between the AT and the switch
	SW[ ]	traffic at the switch
	LATA[ ]	traffic at the LATA
	COC[ ]	central office code homings clliholds a clli code
	equip	holds an equipment code
	swclli[ ][ ]	matrix of clli codes for each switch
integer	HAP[ ][ ]	number of high usage trunks between the AT and the POP
	EOLOC[ ][ ]	location of each EO
	ATLOC[ ][ ]	location of each AT
	SWLOC[ ][ ]	location of each switch
	SW_nlata[ ]	number of LATAs that a switch serves
	POPLOC[ ]	location of each POP
	popswitch[ ]	homing for each POP switch
	IMT[ ][ ]	matrix of backbone trunk traffic

Purpose to read in all live network variables

Called By main()

Calls To none

Local Variables

character	line[200]	
integer	cap	trunk size capacity
	dummy	dummy variable
	pop_length	POP length
	cap_tot	total IEC capacity
	i,j,k,l,m,n,p	loop variables

Global Variables

integer	NUMEO	number of end offices
	NUMAT	number of access tandems
	NUMSW	number of switches

ATPOPSIZE	the number of trunks between the AT and the POP

Algorithmic Description

This function reads in the live network variables from several input files. The input is performed in the following sequence:

1) Read in LEC data from the IEC EO file which contains homing chains from EO-SW and count number of EOs.

2) Read in LEC data from the IEC AT (HAP) file which describes AT-POP homings, which may be multiple.

3) Read in EO V-H coordinates from the EO file.

4) Read in AT V-H coordinates from the AT file.

5) Read in POP V-H coordinates from the IEC POP file. Also read in POP switch homing.

6) Read in Switch V-H coordinates from the IEC Switch file. Also read in and store the CLLI code and the number of LATAs that a switch serves

7) Read the undamaged IMT matrix from qlink. This is the undamaged matrix of IMT capacity. Divide each one-way TG into 50% i->j and 50% j->i to account for bi-directional traffic.

Inputs none

Outputs

returns none

Purpose Before writing network variables to the final output file, we perform the final major task of MCILIVE: creation of the backbone traffic distribution. This matrix helps us know how to handle traffic once it travels up from the LEC networks to the IEC toll switch. Specifically, it determines, for a given switch 'i', what proportion of its access traffic goes to each other remote backbone switch 'j'.

Called By `main()`

Calls To none

Local Variables

integer	`i,j`	loop count variable
	`num_c11=0`	number of class1 to class1 switch pairs
	`num_c13=0`	number of class1 to class3 switch pairs
	`num_c33=0`	number of class3 to class3 switch pairs
	`num_nodirect=0`	with no direct connections
	`cap_c11=0`	capacity of class1 to class1 switch pairs
	`cap_c13=0`	capacity of class1 to class3 switch pairs
	`cap_c33=0`	capacity of class3 to class3 switch pairs
	`iter_flag = 0`	iteration flag
	`num_iter = 1`	number of iterations
real	`vdiff`	vertical difference
	`hdiff`	horizontal difference
	`dist`	distance between switches of interest
	`weight`	a weighting factor
	`Sum[ ]`	sum of access traffic of all switches
character	`line[ 100]`	loop count variable

structure `sw` of type `sw_struct`
with fields:

integer	`class`	switch class
	`cl1_idx`	index of the class1 switch, from 1,n
character	`clli`	switch clli code
	`class1`	switch class

structure `Tij[ ][ ]` of type Tij_struct
with fields:

real	`traf`	traffic 'i' to 'j'
integer	`done`	traffic flag

real	`sumTij[ ]`	sum of traffic 'i' to 'j'
	`offd_traf[ ]`	offered traffic
	`sum_others[ ]`	sum of other traffic
	`egr_traf[ ]`	egress traffic

	`min_prop = 0`	minimum proportion
	`min_test = 0`	minimum test value
	`traf_diff = 0`	traffic difference
	`max_Tij= 0`	maximum traffic 'i' to 'j'

Global Variables

character	`swclli[ ][ ]`	matrix of clli codes for each switch
integer	`NUMSW=0`	number of switches
real	`IMTTraf[ ][ ]`	traffic between switches
	`ACSiec[ ]`	IEC access capacity at the switch
	`SWLOC[ ][ ]`	location of each SW

Algorithmic Description

The results of `TrafDist` go in their own output file, `'trafdist.mci'`. All other network variables are written to the master output file, `'mci2lecam.step1'`, which has a multi-section format comprising some header information, and a section for each network variable. The format and contents of this file are described in more detail in the `OUTWRITE()` function description.

`TrafDist` creates a switch-to-switch traffic distribution file to replace the homomorphism. It outputs this file to `'trafdist.mci'` in the current directory, using the same format as the homomorphism file, so that subsequent code (`mciwdmg`) does not have to be altered to read in a different format.

Step I

In this step, a first approximation of the Tij switch-to-switch matrix is developed. The algorithm first populates Tij pairs for which direct trunk groups exist. (Note: the only Tij pairs are those between class 3 switches and two class 1 switches that serve LEC traffic directly). The Tij pairs are populated with the P.01 engineered load, (since MCI trunks are one-way, this doesn't need to be divided by 2). The Tij entry for the class 1 switches is further divided by 3, since by inspecting the data, these two class 1 switches (i,j) need about 1/3 of their IMTs to serve their own originating traffic. Since direct trunk groups account for 75% of MCI's backbone capacity, most of the proportions are calculated this way. Traffic between the remaining 25% of the switch pairs is estimated by using the proportions of access traffic * 0.3, e.g.;

Tij = 0.3 * offd(i) * offd(j)/(sum of offd(k) for all k != i)

where offd(i) = offered traffic of the switch

The 0.3 weighting signifies that since the switch pair was not important enough to have direct trunking, it probably shares less traffic than the average switch pair, which would have a weighting of 1. Since it is safe to assume that switch pairs without direct trunking are in the bottom 25% and probably have little community of interest, than any weighting between 0.2 and 0.5 would suffice to estimate this.

Step II:

Once the matrix is estimated, the access traffic for each switch is summed from the individual Tij elements. The goal of Part II is to normalize this `sumTij` to `offd_traf`, the offered traffic of the switch. `Offd_traf` for switch i is:

Offd_traf[i] = ACSlec[i] * (1 - PropIntra[i])

Since MCI's IMT trunking is much greater than its LEC access traffic, this is a significant step. An iterative approach is employed, whereby a switch whose `sumTij` estimate is greater than offd_traf systematically "negotiates" with all other switches who have the same problem. For example, if switch i and j both have `sumTij` greater than their respective `offd_traf's`, they decrease their Tij/Tji pairs by the average of (`offd_traf[ i] /SumTij[ i]`) and `offd_traf[ j] /sumTij[ j]`). Thus each switch decreases its traffic by the same amount, avoiding cases where Tij>>Tji, or vice versa. This process is also done for cases where `sumTij < offd_traf`, and iterations continue until every `sumTij` is within 10% of its `offd_traf` or the max of 100 iterations is performed.

Step III:
At this point, we ensure that `offd_traf = egr_traf` at a switch:

*Suppose switch i is sending traffic to a switch j, which is getting too much `egr_traf`. Then Tij is reduces to Tij * (`offd_traf[j]/egr_traf[j]`), and the excess = Tij * (1 - `offd/egr`) is redistributed to all other switches proportionally. The same holds true for switches that don't get enough egress traffic.*

Closing comments: When this algorithm converges, two things should be true:

1. **`sumTij` nearly equal to `offd_traf` for all switches**
2. **`offd_traf` nearly equal to `egr_traf` for all switches**

It may be the case that Tij's for some pairs exceed their direct trunking by as much as two times, which is far off from the original estimate of P.01. However, the algorithm works because MCI's IMT capacity is >> LEC access traffic to MCI, and so there is a good deal of room for these cases to overflow their traffic safely through hierarchical overflow routes without causing blockage. In fact it should be stressed that this algorithm is possible only because of this safety margin, so that a rough approximation that satisfies (1) and (2) above is all that is necessary. The other aspect that makes this algorithm work is that it starts with MCI's own backbone trunking, and only alters traffic patterns proportionately and evenly in each step. Thus, the traffic patterns suggested by the direct trunk groups should be preserved within an order of magnitude.

Presented below is a step by step description of the above algorithm. After declaring and initializing local declarations the following steps are executed.

Preparation for **Step I**

1) Open and read switch homing file. This file gives the homing for class 1 switches for each class 3 switch. A class 1 switch has a homing equal to a blank string.

2) Set class attribute of switch to **Type 1** or **Type 3**

3) Compare order of `sw[ i] .clli` and `swclli[ i]` , as a cross-check

4) Load clli index field

5) Analyze trunking from qlink file (`IMT`).
{This code was developed as a diagnostic of MCI's backbone, and is not necessary for the `TrafDist` algorithm that follows. It determines the proportion of MCI's backbone capacity used for directly connected class 3 switch pairs}.

6) Keep track of switch pairs that are not directly connected.

7) Print resulting summary stats

STEP I `TrafDist` algorithm starts here

1) Populate Tij matrix with traffic

2) If **Type 1** (direct trunking exists):
Then size traffic to P.01 over trunk `IMT[ i][ j]` . Check for class 1 serving LEC traffic and adjust Tij in this case.

3) If **Type 2** (i.e., no direct trunking exists):
Then compute opposite direction, as long as it doesn't have a direct trunk group either.
Report results

STEP II

1) Iterate until offered traffic (`offd_traf`) and `sumTij` agree.

3) Use handshaking to reduce `sumTij` towards `offd_traf`.

4) Look for another switch that also has a value of `sumTij` that is to high, so that switches i and j can mutually lower traffic to each other by the same proportion.

5) Calculate proportion for switch i and j, and use average as proportion for mutually reducing Tij/Tji.

6) Discount old Tij/Tji from sums:
```
sumTij[ i]  -= Tij[ i][ j] .traf
sumTij[ j]  -= Tij[ j][ i] .traf.
```

7) Calculate new Tij/Tji as proportionately smaller:
```
Tij[ i][ j] .traf = min_prop * Tij[ i][ j] .traf
Tij[ j][ i] .traf = min_prop * Tij[ j][ i] .traf.
```

8) Impose minimum of 1 erlang between each pair:
```
if (Tij[ i][ j] .traf<1.0) Tij[ i][ j] .traf=1.0
if (Tie[ j][ I] .tram<1.0) Tie[ j][ I] .traf=1.0.
```

9) Add new Tij/Tji into sums, so that sums are kept up to date along the way:
```
sumTij[ i]  += Tij[ i][ j] .traf
sumTij[ j]  += Tij[ j][ i] .traf.
```

10) Use handshaking to increase `sumTij` look for another switch that also has `sumTij` to high, so that switches i and j can mutually lower traffic to each other by the same proportion.

11) Calculate proportion for switch i and j, and use average as proportion for mutually reducing Tij/Tji
```
min_prop = offd_traf[ i] /sumTij[ i]
min_test = offd_traf[ j] /sumTij[ j]
min_prop = (min_test + min_prop)/2.0.
```

12) Discount old Tij/Tji from sums
```
sumTij[ i]  -= Tij[ i][ j] .traf
sumTij[ j]  -= Tij[ j][ i] .traf.
```

13) Calculate new Tij/Tji as proportionately smaller
```
Tij[ i][ j] .traf = min_prop * Tij[ i][ j] .traf.
Tij[ j][ i] .traf = min_prop * Tij[ j][ i] .traf.
```

14) Impose minimum of 1 erlang between each pair
```
if (Tij[ i][ j] .traf<1.0) Tij[ i][ j] .traf=1.0
if (Tij[ j][ i] .traf<1.0) Tij[ j][ i] .traf=1.0.
```

15) Add new Tij/Tji into sums, so that sums are kept up to date along the way.
```
sumTij[ i]  += Tij[ i][ j] .traf
sumTij[ j]  += Tij[ j][ i] .traf.
```

16) Use handshaking to increase `sumTij`

17) Test Convergence
If `sumTij` is off by more than 10% for any switch, don't converge; otherwise, set flag to converge. Print progress for each iteration as the total amount of traffic difference between the `sumTij`'s and the `offd_traf's`.

18) Set up for **Step III**.
Calculate Egress Traffic and print intermediate results.

STEP III

1) Perform `offd_traf<==>egr_traf` equalization. {Note that this step does not use handshaking, so that decreases/increases in Tij are not mirrored for Tji direction. This is why this step is done last and only once}.

2) Increase or decrease allocation from i to j, depending on j's balance of `offd/egr` traffic adjust `Tij[ i][ j] .traf` proportionately to `offd/egr` ratio.

3) Impose minimum of 1 erlang between each pair.

4) Renormalize Tij so that it sums to `offd_traf.`

5) Impose minimum of 1 erlang between each pair

6) Resum `sumTij` from Tij now that it's normalized.

7) Calculate Egress Traffic and print final results

8) Print Output `TrafDist` File

Inputs none

Outputs please see description of output file "`mci2lecam file`" under the module description section.

Purpose to format and print data for input to lecam.c

Called By `main()`

Calls To none

Local Variables

integer	`count, k, l, m, n`	loop count variables

Global Variables

integer	NUMEO	number of end offices
	NUMAT	number of access tandems
	NUMPOP	number of points of presence
	NUMSW	number of switches

Algorithmic Description This function writes out all output variables to the output file " `mci2lecam.step1`" for a list of these variables and the file format see the description of the output file under the module description section.

3.3 sprlive: Build Sprint Live Network and Traffic

Purpose

The purpose of this module is to complete the representation of the undamaged Sprint network. This modules uses existing publicly available data on that network along with telco engineering and reverse engineering methods to completely represent the Sprint network from the TGs down to the LECS.

This program also allocates traffic from the switches down to the EOs, creating traffic patterns along the way on EO-POP, AT-POP, and EO-AT TGs and builds a traffic matrix. This module is used in conjunction with `sprwdmg` which introduces damage effects into the network data files. The final output of the two step process after the `sprwdmg` module is input into the LECAM module.

Call Syntax

`sprlive -k <key file> [ mandatories][ options]`

mandatory:	*special syntax:*	*function*:
`-k`	`<keyfile>`	reads in input file `<key file>`
optional;	*special syntax:*	*function:*
`-o`	`<outfile>`	reads in output file name. If not specified, default file name is `spr2lecam.step1`
`-t`	`<#>`	invokes `Trafdist()`, traffic distribution matrix routine, default setting
`-h`		invokes new High Usage occupancy and overflow algorithm commands (this setting is now considered the default setting)
`-p`		High Usage overflow default setting override
`-?`		user help--prints call syntax and exits without running

example `sprlive -k dplive.key -h` (spaces optional)

Input Files

`key file` This file contains hardcoded instructions for opening up and reading in the various data files to be used in this module.

format **SEE SECTION 2.1/EXHIBIT 2-3**

Output Files

`spr2lecam file`

This file contains the formatted output data to be used in the LECAM module

format
header:

* SPRINT live data file (damage added in MC loop)
<IEC label>
("IECid=", c3)
<switch size>
("SWITCH_SIZE", i3)
<POP size>
("POP_SIZE", i3)
<AT size>
("AT_SIZE", i4)
<EO size>
("EO_SIZE", i5)

<AT to POP size>
("ATPOPSIZE", i3)

EOAT
section 1: <traffic at the AT for each EO>,
(i4, I4,.....) in rows of 20

HUPOP
section 2: <traffic between high usage AT and the POP for each EO>,
(i4, I4,.....) in rows of 20

ATPOP
section 3: <traffic between AT and the POP for each EO>,
(i4, I4,.....) in rows of 20

HAP
section 4: <number of high usage trunks AT and the POP for each EO>,
(i4, I4,.....) in rows of 20

SDlive
section 5: < size of trunks between EO and POP>,
(i5, I5,.....) in rows of 15

TEP
section 6: <traffic between EO and the POP for each EO>,
(f8.2, f8.2, ...) in rows of 10

TAP
section 7: <traffic between AT and the POP for each EO>,
(f8.2, f8.2, ...) in rows of 10

F0
section 8: <number of high usage trunks AT and the POP for each EO>,
(f14.8, f14.8, ...) in rows of 2

AClive
section 9: <total access capacity>,
(f.8, f.8, ...) in rows of 5

ACSlive
section 10: < total access capacity for the LEC>,
(f.8, f.8, ...) in rows of 5

ELAP
section 11: <traffic between AT and the POP for each AT>,
(f8.2, f8.2, ...) in rows of 6

SAPlive
section 12: <size of trunks between AT and the POP for each AT>,
(i5, I5,.....) in rows of 6

D3
section 13: < access traffic engineered from LEC switches to POP>,
(f8.2, f8.2, ...) in rows of 10

ACSPOP
section 14: < access traffic engineered from IEC switches to POP>,
(f8.2, f8.2, ...) in rows of 10

POPSWITCH
section 15: < POP switch homing>,
(i4, i4, ...) in rows of 20

EOlata
section 16: < LATA each EO is homed to >,
(i4, i4, i4, ...) in rows of 20

PhysDiv
section 17: <physical diversity value for each end office: 0/1>,
(i1, i1, ...) in rows of 80

IMT
section 18: < matrix of backbone trunk traffic for each switch>,
(i5, i5, ...) in rows of 15

PropIntra
section 19: < proportion of intraswitch traffic for each switch>,
(f.8, f.8, ...) in rows of 5

Includes

"sprlive.h"	Defines global network and traffic variables and constants
"lecam.h"	Defines network sizing constants
"fileio.c"	User-defined I/O functions; see Appendix A

Constants

Constants used in sprlive and defined in "sprlive.h":

OVFLW	0.90	Sprint overflows 10% from HU to final
TRKOCC	0.95	HUs are 95% occupied
INTRA	0.0	multiplier for interswitch traffic to add intra
MAX_ITER	100	maximum number of iterations for traffic algorithm
TOLERANCE	0.1	precision level

Constants used in sprlive and defined in "lecam.h":

TRUE	1	true
FALSE	0	false
MAX_AT	1000	maximum number of access tandems
MAX_EO	19250	maximum number of end offices
ATPOPMAX	9	maximum number of POPs a single AT may home to
MAX_POP	575	maximum number of points of presence
MAX_SWITCH	130	maximum number of switches
MAX_LATA	1000	maximum number of LATAs
MAX_REGION	5	maximum number of regions covered by a LATA

Global Variables

global variables defined in "sprlive.h"

integer	KEYTOG	toggle to indicate input 'file name' was read in successfully
	NUMEO	number of end offices
	NUMAT	number of access tandems
	NUMPOP	number of points of presence
	NUMSW	number of switches
	ATPOPSIZE	the number of trunks between the AT and the POP
	HAP[MAX_AT] [ATPOPMAX]	number of high usage trunks between the AT and the POP
	AT[MAX_EO]	traffic at the AT
	ATPOP[MAX_EO]	traffic between the AT and the POP
	HUPOP[MAX_EO]	traffic between the high usage AT and the POP
	TYPE[MAX_EO]	type of homing between the AT and the switch
	SW[MAX_EO]	traffic at the switch
	LATA[MAX_EO]	traffic at the LATA
	COC[MAX_EO]	central office code homings
	IMT[MAX_SWITCH] [MAX_SWITCH]	matrix of backbone trunk traffic
	SEP[MAX_EO]	size of trunks between EO and POP
	SAP[MAX_AT] [ATPOPMAX]	size of trunks between AT and POP
	COCSW[MAX_SWITCH]	matrix of homings between switches and COCs
	PhysDiv[MAX_EO]	physical diversity value for each EO of TYPE = 3
	SW_nlata[MAX_SWITCH]	number of LATAs that a switch serves
	popswitch[MAX_POP]	POP switch homing

	D3DONE[MAX_AT] [ATPOPMAX]	a logical control variable to avoid double counting AT to POP trunks groups
	TOT_IMT	total number of backbone trunks at the switch
	EOLOC[MAX_EO] [2]	location of each EO
	ATLOC[MAX_AT] [2]	location of each AT
	SWLOC[MAX_SWITCH] [2]	location of each switch
	POPLOC[MAX_POP] [2]	location of each POP
	EOA	originating end office
	EOB	terminating end office
	TOTAT	total traffic at the AT
	TOTEO	total traffic at the EO
	TOTSW	total traffic at the switch
	TOTT2	total traffic of Type 2
	TOTT3	total traffic of Type 3
real	IMTTraf[MAX_SWITCH] [MAX_SWITCH]	traffic between switches
	TEP[MAX_EO]	traffic EO to POP
	TEA[MAX_EO]	traffic EO to AT
	TAP[MAX_AT] [ATPOPMAX]	traffic AT to POP
	ACSlec[MAX_SWITCH]	LEC access capacity at the switch. This is the sum of the D3[POP]s for POPs homed to SW. This is a function of TEP and TAP. Alternatively, ACSlec is the sum of AC[k] for all EOs k homed to SW. These two methods should yield the same answer (within rounding). This is computed before damage.
	ACS[SW]	This parameter is derived by summing the P.01 engineered traffic on IMTs going into SW, times 0.5 for one-way. This parameter is later decremented in the allocation algorithm, and is thus unsuitable for the calculationsthat follow that algorithm. ACS is a function of IMTTraf.
	ACSiec[MAX_SWITCH]	Exact copy of ACS[SW] , but is not decremented in the allocation algorithm. ACSiec is used to compute ACSPOP. ACSiec is a function of IMTTraf.
	AC[MAX_EO]	total access capacity
	D3[MAX_POP]	This is the access traffic arriving at POP from the EO-POP and AT-POP TGs homed to it. D3 is a function of TEP and TAP. This is equal to intraswitch plus interswitch traffic at a POP
	ACSPOP[POP]	This is the access traffic that POP can offer to SW for transmission across IMTs. This is D3 * (ACSiec/ACSlec), and is written to outfile as the "real" D3, since it more accurately reflects engineered load at a POP. ACSPOP is a function of IMTTraf. It is computed before D3 is decremented for damage, so that it reflects true all-alive engineered traffic. It is equal to interswitch traffic (only) at a POP.
	ACSlec[SW]	This is the sum of the D3[POP] s for POPs homed to SW. This is a function of TEP and TAP. Alternatively, ACSlec is the sum of AC[k] for all EOs k homed to SW. This is computed before damage.

	ACPiec[MAX_POP]	access traffic engineered from IEC switches to POP
	PropIntra[MAX_SWITCH]	proportion of intraswitch traffic
	fProp[MAX_SWITCH]	ratio intraswitch traffic to interswitch traffic
	Overflow[MAX_EO]	traffic overflowing AT to POP on high usage trunks
	TOTFO	total focused overload traffic
double	FO[MAX_EO][2]	divided into HU traffic (0) and FINAL traffic (1)
character	swclli[MAX_SWITCH][12]	matrix of clli codes for each switch
	clli[12]	individual clli code
	equip[4]	four character equipment code

global variables defined in "keyfiles.h"

FILE	*fileptr	points to an open input file
	*outfile	points to an open output file
	*fptr	points to an open output file
character	eo_file[80]	name of the EO file
	at_file[80]	name of the AT file
	iec_eo_file[80]	name of the IEC EO file
	iec_at_file[80]	name of the IEC AT file
	iec_sw_file[80]	name of the IEC SW file
	iec_pop_file[80]	name of the IEC POP file
	morph_file[80]	name of the morphing file
	swhmg_file[80]	name of the switch homing file

Local Variables

Variables local to main()

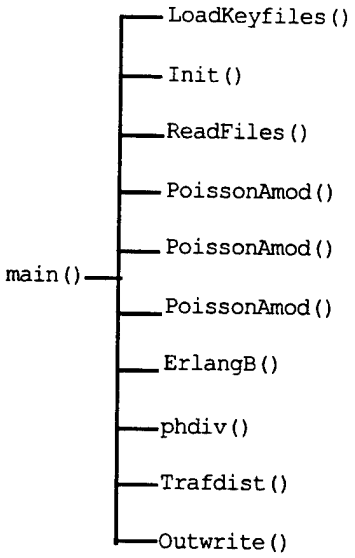
extern	character	*optarg	a string containing a single command line argument.
integer		optind	the number of single command line arguments to be processed, supplied externally by the operating system
character	c		a command line option character
	outfile[80]		the name of the output file
	line[200]		buffer for parsing a single input line from a file
	livekeyfile[80]		the name of the input key file for the live networks
	**argv[ ]		an array containing all of the command line arguments
integer	err=0		error flag indicating a problem with the command line arguments
	otog=0		indicates use of the [-o] option (a user supplied output file name)
	ttog=0		command line parameter setting indicates the use of the [-t] option (currently not used)
	htog=0		indicates the [-h] option (use of new high usage rules, this is now the default setting)
	ptog=0		indicates the [-p] option (override use of new high usage overflow rules)
	test		not used

	`ovflw_count=0`	the number of end offices that overflow from HU to final homing (same as `t3_num`)
	`dummy`	used to read in a dummy integer value
	`cap`	total capacity
	`i,j,k,l,m,n,p,pm`	loop count variables
	`SEP_total=0`	sums the number of high usage trunks between EOs and POPs
	`argc`	the number of command line arguments
structure	`AT_hmg[ MAX_AT]` of type `AT_hmg_struct`	
	with fields:	
integer	`type2`	number of Type 2 homing switches
	`type3`	number of Type 3 homing switches
real	`ovflw_total=0`	the total amount of traffic which overflows from the high usage trunks
	`TEP_total=0`	the total amount of traffic between EOs and POPs
	`TEA_total=0`	the total amount of traffic between EOs and ATs
	`wovflw_total=0`	the weighted overflow based on access capacity across all EOs
	`wovflw_denom=0`	the total access capacity across all EOs
	`sum2_AC=0`	sum of the access capacity for Type 2 switches
	`sum3_AC=0`	sum of the access capacity for Type 3 switches

Component Functions

void	`Init()`	intializes all network variables
	`ReadFiles()`	reads in each section of the input files specified in the `<keyfile>`
	`Trafdist()`	This routine defineS the community of interest weighting that describes how much or how little traffic a switch will send to another switch due to their proximity.
	`OutWrite()`	writes out each section of the output file
	defined in "`sprlive.h`"	
void	`LoadKeyfile()`	loads input file names from the `<keyfile>` specified in the command line
double	`ErlangB()`	traffic calculator for engineering overflow capacity using ErlangB blocking
	`PoissonA()`	traffic calculator for for engineering traffic from switchA to switchB and vice-versa
	`PoissonAmod()`	traffic calculator for engineering traffic from switchA to switchB and vice-versa
	`PoissonNmod()`	traffic calculator for number of trunks
integer	`PoissonN(Ain,Bin)`	traffic calculator for number of trunks
	`phdiv(eo,at,pop)`	determines physical diversity based on co-location

Function Tree



Algorithmic Description

This module completes the representation of the Sprint network and the traffic that is carried on that network. The procedure is to distribute the Sprint traffic down though the local exchange carriers (LEC) in order to build a reasonable model of the network and its relationship to the LECs. A similar process is undertaken with AT&T and Sprint.

Sprint provides the sizes of High Usage trunks (HU) and access tandems (AT) to point-of-presence (POP) trunk groups (TG), which are used to engineer traffic in the LECs. Where homings through an AT are involved, traffic must be allocated down from the AT to the end offices (EO). There are several ways to do this, each with drawbacks and advantages. This module uses the most accurate method where possible; cases where this method breaks down are detected, and a secondary method is used to allocate traffic from these ATs down to their EOs.

The `main()` routine contains the majority of the algorithmic code for this module. After the command line toggles have been interpreted, the `main()` routine begins by passing the `<keyfile>` argument into `LoadKeyfile()`, which reads in the names of the LEC and interexchange (IEC) files. It then calls `Init()`, which initializes all network variables. `Readfile()` is called next to read in all network data. `Main()` then opens the `<outfile>` and proceeds to perform the core of the algorithmic work in this module, in the following order:

1) Call `PoissonAmod()` which uses the number of backbone/IEC trunks to compute the backbone traffic across the trunks. Traffic i --> j is 1/2 load of the combined 2 way trunk group.

2) Engineer total interswitch traffic at each switch by summing traffic from all backbone trunks that end at a switch.
   Set `access traffic` = to one half of total traffic

3) Use a **3-Pass** algorithm to allocate traffic to individual LEC network segments, for 3 Types of LEC Trunk group size homings: **Type 1**, HU-only homings, **Type 2**,

AT-only homings, and **Type 3**, homings that involve an HU overflowing to the AT (final) homing :

In general calculate:

- AC (access traffic offered by an EO)
- TEA (traffic going on the EO-AT TG, if any)
- TAP (traffic going on the AT-POP TG, if any)
- TEP (traffic going on the EO-POP TG, if any).

PASS 1 Engineer the straightforward cases and set up statistics to support the more complex cases.

Type 1. HU-only homing.
Call `PoissonAmod()` to engineer AC directly from the EO-POP TG size, assuming P.01, (1% blocking).
Set EO-POP traffic equal to AC.
De-allocate EO AC from switch (SW) traffic.

Type 2. AT-only homing.
Traffic over AT-only homings cannot be allocated until the effect of overflow from HU trunks is determined. In the first pass we sum up central office codes (COC) for all EOs homed to the same AT-POP TG.

Type 3. HU with AT overflow.
Traffic for HU homings that overflow through the AT is calculated using the methodology shown below. In later passes, it is determined if this method broke down for a particular AT. If so, an alternative algorithm is used.

The preferred algorithm is:

A) Call `PoissonAmod()` to engineer communications sent directly EO-POP (with P.80 blocking on the HU TG meaning the last trunk in the TG is occupied 80% of the time).

B) Call `ErlangB()` to engineer overflow capacity sent first through the AT then to the SW. Traffic overflowing to the AT is computed by offering AC to the HU TG using Erlang B blocking. This Erlang B blockage is used to determine traffic carried on the HU, and traffic that must overflow from the EO to the AT.

C) Compute the running sum of EO-AT traffic for each AT-POP TG to determine if our methodology would create more AT-POP traffic than could be handled by the AT-POP TG. As a precautionary measure, compute the variables required by the backup algorithm: the sum of COCs and overflow-weighted COCs to each AT-POP TG for the **Type 3** homings.

PASS 1 .5 This pass only applies to **Type 3** EOs.
No network variables are calculated. Only determines if first-choice numbers in previous pass are valid.

Type 1. HU-only homing. No calculations for this pass.

Type 2. AT-only homing. No calculations for this pass.

Type 3. HU with AT overflow.
Determines whether the first or second-choice algorithm will be used for allocating AT-POP traffic down to EOs. If the first-choice algorithm won't work (i.e. causes AT-POP traffic to be greater than what the AT-POP TG can accommodate), then a flag is set for the next pass, and the **Type 3** overflow-weighted COCs are combined with the **Type 2** COCs to create the total needed by the alternative algorithm.

If the first choice algorithm will work, then the share of AT-POP traffic already allocated to **Type 3** EOs is subtracted from the amount available to be allocated to the **Type 2** EOs that use the same AT.

PASS 2 The second pass allocates traffic to the **Type 2** EOs (AT-only homings) regardless of the method used for **Type 3** traffic allocation. Calculate traffic the **Type 3** EOs for the cases where the alternative allocation algorithm must be used.

Type 1. HU-only homing. No calculations for this pass.

Type 2. AT-only homing.
Use COCs and the AT-POP traffic to engineer the access capacity and the EO-AT traffic and then to the SW.
Calculate AC based on COCs
Allocate traffic AT-POP
Calculate traffic EO-AT
De-allocate AC traffic at the SW

Type 3. HU with AT overflow.
Use second-choice algorithm:

If htog is set to true
Allocate AT-POP traffic based on overflow weighted COCs
Replace previous values of `TEA` and `AC`
De-allocate `AC` traffic at the SW
else
Allocate AT-POP traffic based on HU overflow weighted COCs
De-allocate AC traffic at the SW
Replace previous values of EO-AT traffic and AC.

At this point all LEC network traffic variables have been engineered and the calculation of Traffic and the determination of Trunks is complete in the LEC .

4) Reduce AC by 1/2 to capture access capacity only, assuming access = egress at an EO. Perform this for each EO in the LEC and for each SW in the IEC.
Sum traffic from LECs up to SW (`ACSlec`).

5) For each **Type 3** homing (HU with AT overflow), call `phdiv()` determine physical diversity. (i.e. collocated = not physically diverse; not collocated = physically diverse). This is necessary to determine if physical damage to an AT site will also cause damage to the HUE homing.

6) Print `ACS, ASCII` and `ACSlec` for each SW

7) Compute and print overflow and occupancy summary statistics:
Sum AC for each **Type 2.**
Sum **Type 2** homings.
Verify actual engineered overflow form the HU TG to the POP.
Sum AC for each **Type 3.**
Calculate average and weighted average overflow.
Calculate EO-POP and EO-AT traffic.

8) Use `ACSlec` and results of `phdiv()` to recalculate `SW AC`. Backbone trunks that connect to a SW with no EO homings should not be included in calculating `SW AC`. This occurs when a SW is collocated.

9) Calculate AT traffic engineered between LEC switches and POP.
Sum HU traffic up to POPs.

Sum AT traffic up to POPs

Switch and Pop variables are now calculated, including versions that include inter-switch traffic only, and versions that include both inter- and intraswitch traffic. From this, the proportion of intraswitch traffic at each switch (`PropIntra`) is calculated.

10) Calculate the amount of inter-switch access traffic at a POP. Determine iintraswitch traffic.

11) Calculate the proportion of intraswitch traffic and set the lowest value of `PropIntra` = 0.3

12) Calculate one-half the value of : traffic EO to POP and traffic EO to AT.

    At this point the calculation of Traffic and the determination of Trunks is completed in the LEC

13) If `ttog` set on then call `TrafDist()`. This routine defines the community of interest weighting that describes how much or how little traffic a switch will send to another switch due to their proximity.

14) Call `OutWrite()` which formats and writes all variable to the outfile which is a LECAM input file.

Inputs none; operates on global variables

Outputs

global	integer	NUMEO=0	number of end offices
		NUMAT=0	number of access tandems
		NUMPOP=0	number of points of presence
		NUMSW=0	number of switches
		TOT_IMT=0	total number of backbone trunks at the switch
		AT[i] =0	traffic at the AT
		ATPOP[i] =0	traffic between the AT and the POP
		HUPOP[i] =0	traffic between the high usage AT and the POP
		TYPE[i] =0	type of homing between the AT and the SW
		SW[i] =0	traffic at the switch
		LATA[i] =0	LATA each EO is homed to
		COC[i] =0	central office code homings
		SEP[i] =0	size of trunks between EO and POP
		PhysDiv[i] =0	physical diversity value for each EO of TYPE = 3
		F0[i][0] =0	engineered traffic offered at an EO
		F0[i][1] =0	engineered traffic offered at an EO
		Overflow[i] =0	traffic overflowing AT to POP on high usage trunks
		D3DONE[i][j] =0	a logical control variable to avoid double counting AT to POP trunks groups
		HAP[i][j] =0	number of high usage trunks between the AT and the POP
		SAP[i][j] =0	size of trunks between AT and POP
		fProp[i] =0	ratio intraswitch traffic to interswitch traffic
		SW_nlata[i] =0	number of LATAs that a switch serves
		COCSW[i] =0	matrix of homings between switches and COCs
		ACSlec[i] =0	LEC access capacity at the switch
		ACSiec[i] =0	IEC access capacity at the switch
		PropIntra[i] =0	proportion of intraswitch traffic
		IMT[i][j] =0	matrix of backbone trunk traffic
global	real	TEP[i] =0.0	traffic EO to POP
		TEA[i] =0.0	traffic EO to AT
		AC[i] =0.0	total access capacity
		TAP[i][j] =0.0	traffic AT to POP
		D3[i] =0.0	access traffic engineered from LEC switches to POP
		ACPiec[i] =0.0	access traffic engineered from IEC switches to POP
		IMTTraf[i][j] =0.0	traffic between switches

returns no formal returns

Purpose Initializes all network variables

Called By main()

Calls To none

Local Variables

integer	`i, j`	loop count variables

Global Variables

none

Global Constants

`MAX_EO`	maximum number of end offices
`MAX_AT`	maximum number of access tandems
`ATPOPSIZE`	the number of trunks between the AT and the POP
`MAX_POP`	maximum number of points of presence
`MAX_SWITCH`	maximum number of switches

Algorithmic Description

For every EO this function sets the trunk sizes and traffic carried between EOs, ATs, POPs, SWs to initial values (0) and does other things for ATs, POPs, SWs and traffic in backbone. Initializes all network variables

Inputs

file	iec_eo_file	IEC EO file
	iec_at_file	IEC AT file
	iec_pop_file	IEC POP file
	qlink_file	QTCM link file
	eo_file	EO file
	at_file	AT file

Outputs

character	AT[ ]	traffic at the AT
	ATPOP[ ]	traffic between the AT and the POP
	HUPOP[ ]	traffic between the high usage AT and the POP
	TYPE[ ]	type of homing between the AT and the SW
	SW[ ]	traffic at the switch
	LATA[ ]	traffic at the LATA
	COC[ ]	central office code homings
	clli	holds a clli code
	equip	holds an equipment code
	swclli[ ][ ]	matrix of clli codes for each switch
integer	HAP[ ][ ]	number of high usage trunks between the AT and the POP
	EOLOC[ ][ ]	location of each EO
	ATLOC[ ][ ]	location of each AT
	SWLOC[ ][ ]	location of each Switch
	SW_nlata[ ]	number of LATAs that a switch serves
	POPLOC[ ]	location of each POP
	popswitch[ ]	homing for each pop switch
	IMT[ ][ ]	matrix of backbone trunk traffic

Purpose to read in all live network variables

Called By main()

Calls To none

Local Variables

character	line[200]	
integer	cap	trunks size capacity
	dummy	dummy variable
	pop_length	POP length
	cap_iec_tot	total IEC capacity
	i,j,k,l,m,n,p	loop variables

Global Variables

integer	NUMEO	number of end offices
	NUMAT	number of access tandems

NUMSW	number of switches
ATPOPSIZE	the number of trunks between the AT and the POP

Algorithmic Description

This function reads in the live network variables from several input files. The input is performed in the following sequence:

1) Read in LEC data from IEC EO file which contains homing chains from EO-SW and count number of EOs

2) Read in LEC data from the IEC AT (HAP) file which describes AT-POP homings, which may be multiple.

3) Read in EO V-H coordinates from the EO file.

4) Read in AT V-H coordinates from the AT file.

5) Read in Switch V-H coordinates from the IEC Switch file. Also read in and store the CLLI code and the number of LATAs that a switch serves

6) Read in POP V-H coordinates from the IEC POP file. Also read in POP switch homing.

7) Read the undamaged IMT matrix from the qlink file.

Inputs none

Outputs

returns none

Purpose Before writing network variables to the final output file, we perform the final major task of SprintLIVE: creation of the backbone traffic distribution. This matrix helps us know how to handle traffic once it travels up from the LEC networks to the IEC toll switch. Specifically, it determines, for a given switch 'i', what proportion of its access traffic goes to each other distant backbone switch 'j'.

Called By `main()`

Calls To none

Local Variables

integer	`i,j`		loop count variable
	`num_c11=0`		number of class1 to class1 switch pairs
	`num_c13=0`		number of class1 to class3 switch pairs
	`num_c33=0`		number of class3 to class3 switch pairs
	`num_nodirect=0`		with no direct connections
	`cap_c11=0`		capacity of class1 to class1 switch pairs
	`cap_c13=0`		capacity of class1 to class3 switch pairs
	`cap_c33=0`		capacity of class3 to class3 switch pairs
	`iter_flag = 0`		iteration flag
	`num_iter = 1`		number of iterations
real	`vdiff`		vertical difference
	`hdiff`		horizontal difference
	`dist`		distance between switches of interest
	`weight`		a weighting factor
	`Sum[ ]`		sum of access traffic of all switches
character	`line[ 100]`		loop count variable
structure	`sw` of type `sw_struct`		
	with fields:		
	integer	`class`	switch class
		`c11_idx`	index of the class1 switch, from 1,n
	character	`clli`	switch clli code
		`class1`	switch class
structure	`Tij[ ][ ]` of type Tij_struct		
	with fields:		
	`real`	`traf`	traffic 'i' to 'j'
	`integer`	`done`	traffic flag
real	`sumTij[ ]`		sum of traffic 'i' to 'j'
	`offd_traf[ ]`		offered traffic
	`sum_others[ ]`		sum of other traffic

	egr_traf[ ]	egress traffic
	min_prop = 0	minimum proportion
	min_test = 0	minimum test value
	traf_diff = 0	traffic difference
	max_Tij= 0	maximum traffic 'i' to 'j'

Global Variables

character	swclli[ ][ ]	matrix of clli codes for each switch
integer	NUMSW=0	number of switches
real	IMTTraf[ ][ ]	traffic between switches
	ACSiec[ ]	IEC access capacity at the switch
	SWLOC[ ][ ]	location of each SW

Algorithmic Description

The results of `trafdist` go in their own output file, `'trafdist.spr'`. All other network variables are written to the master output file, `'spr2lecam.step1'`, which has a multi-section format comprising some header information, and a section for each network variable. The format and contents of this file are described in more detail in the `OUTWRITE()` function description.

`TrafDist` creates a switch-to-switch traffic distribution file to replace the homomorphism. It outputs this file to `'trafdist.spr'` in the current directory, using the same format as the homomorphism file, so that subsequent code (`sprwdmg`) does not have to be altered to read in a different format.

Step I

In this step, a first approximation of the Tij switch-to-switch matrix is developed. The algorithm first populates Tij pairs for which direct trunk groups exist. (Note: the only Tij pairs are those between class3 switches and two class1's that serve LEC traffic directly). The Tij pairs are populated with the P.01 engineered load, (since Sprint trunks are one-way, this doesn't need to be divided by 2). The Tij entry for the class1 switches is further divided by 3, since by inspecting the data, these two class1 switches (i,j) need about 1/3 of their IMTs to serve their own originating traffic. Since direct trunk groups account for 75% of Sprint's backbone capacity, most of the proportions are calculated this way. Traffic between the remaining 25% of the switch pairs is estimated by using the proportions of access traffic * 0.3, e.g.;

Tij = 0.3 * offd(i) * offd(j)/(sum of offd(k) for all k != i)

where offd(I) = offered traffic of the switch

The 0.3 weighting signifies that since the switch pair was not important enough to have direct trunking, it probably shares less traffic than the average switch pair, which would have a weighting of 1. Since it is safe to assume that switch pairs without direct trunking are in the bottom 25% and probably have little community of interest, then any weighting between 0.2 and 0.5 would suffice to estimate this.

Step II:

Once the matrix is estimated, the access traffic for each switch is summed from the individual Tie elements. The goal of Part II is to normalize this `sumTij` to `offd_traf`, the offered traffic of the switch. `Offd_traf` for switch i is:

Offd_traf[i] = ACSlec[i] * (1 - PropIntra[i])

Since Sprint's IMT trunking is much greater than its LEC access traffic, this is a significant step. An iterative approach is employed, whereby a switch whose `sumTij` estimate is greater than `offd_traf` systematically "negotiates" with all other switches who have the same problem. For example, if switch i and j both have `sumTij` greater than their respective offd_traf's, they decrease their Tij/Tji pairs by the average of (`offd_traf[ i] /sumTij[ i]`) and `offd_traf[ j] /sumTij[ j]`). Thus each switch decrease its traffic by the same amount, avoiding cases where Tij>>Tji, or vice versa. This process is also done for cases where sumTij < `offd_traf`, and iterations continue until every sumTij is within 10% of its `offd_traf` or the max of 100 iterations is performed.

Step III:
At this point, we ensure that offd_traf = egr_traf at a switch:

*Suppose switch i is sending traffic to a switch j, which is getting too much `egr_traf`. Then Tij is reduced to Tij * (`offd_traf[j]/egr_traf[j]`), and the excess = Tij * (1 - `offd/egr`) is redistributed to all other switches proportionally. The same holds true for switches that don't get enough egress traffic.*

Closing comments: When this algorithm converges, two things should be true:

1. **`sumTij` nearly equal to `offd_traf` for all switches**
2. **`offd_traf` nearly equal to `egr_traf` for all switches**

It may be the case that Tij's for some pairs exceed their direct trunking by as much as two times, which is far off from the original estimate of P.01. However, the algorithm works because Sprint's IMT capacity is >> LEC access traffic to Sprint, and so there is a good deal of room for these cases to overflow their traffic safely through hierarchical overflow routes without causing blockage. In fact it should be stressed that this algorithm is possible only because of this safety margin, so that a rough approximate that satisfies (1) and (2) above is all that is necessary. The other aspect that makes this algorithm work is that it starts with Sprint's own backbone trunking, and only alters traffic patterns proportionately and evenly in each step. Thus, the traffic patterns suggested by the direct trunk groups should be preserved within an order of magnitude.

Presented below is a step by step description of the above algorithm. After declaring and initializing local declarations the following steps are executed.

Preparation for **Step I**

1) Open and read switch homing file. This file gives the homing for class1 switches for each class3 switch. A class1 switch has a homing equal to a blank string.

2) Set class attribute of switch to **Type 1** or **Type 3**

3) Compare order of `sw[ i] .clli` and `swclli[ i]` , as a cross-check

4) Load clli index field

5) Analyze trunking from `qlink` file (`IMT`).
{This code was developed as a diagnostic of Sprint's backbone, and is not necessary for the TrafDist algorithm that follows It determines the proportion of Sprint's backbone capacity used for directly connected class3 switch pairs}.

6) Keep track of switch pairs that are not directly connected.
7) Print resulting summary stats

STEP I `TrafDist` algorithm starts here

1) Populate Tij matrix with traffic

2) If **Type 1** (direct trunking exists):
Then size traffic to P.01 over trunk `IMT[ i][ j]` . Check for class1 serving LEC traffic and adjust Tij in this case.

3) If **Type 2** (i.e., no direct trunking exists):
Then Compute opposite direction, as long as it doesn't have a direct trunk group either.
Report Results

STEP II

1) Iterate until offered traffic (`offd_traf`) and sumTij agree.

3) Use handshaking to reduce sumTij towards offd_traf.

4) Look for another switch that also has a value of sumTij that is to high, so that switches i and j can mutually lower traffic to each other by the same proportion.

5) Calculate proportion for switch i and j, and use average as proportion for mutually reducing Tij/Tji.

6) Discount old Tij/Tji from sums:
```
sumTij[ i]  -= Tij[ i][ j] .traf
sumTij[ j]  -= Tij[ j][ i] .traf.
```

7) Calculate new Tij/Tji as proportionately smaller:
```
Tij[ i][ j] .traf = min_prop * Tij[ i][ j] .traf
Tij[ j][ i] .traf = min_prop * Tij[ j][ i] .traf.
```

8) Impose minimum of 1 erlang between each pair:
```
if (Tij[ i][ j] .traf<1.0) Tij[ i][ j] .traf=1.0
if (Tij[ j][ i] .traf<1.0) Tij[ j][ i] .traf=1.0.
```

9) Add new Tij/Tji into sums, so that sums are kept up to date along the way:
```
sumTij[ i]  += Tij[ i][ j] .traf
sumTij[ j]  += Tij[ j][ i] .traf.
```

10) Use handshaking to increase sumTij look for another switch that also has sumTij to high, so that switches i and j can mutually lower traffic to each other by the same proportion.

11) Calculate proportion for switch i and j, and use average as proportion for mutually reducing Tij/Tji
```
min_prop = offd_traf[ i] /sumTij[ i]
min_test = offd_traf[ j] /sumTij[ j]
min_prop = (min_test + min_prop)/2.0.
```

12) Discount old Tij/Tji from sums
```
sumTij[ i]  -= Tij[ i][ j] .traf
sumTij[ j]  -= Tij[ j][ i] .traf.
```

13) Calculate new Tij/Tji as proportionately smaller

```
Tij[ i][ j] .traf = min_prop * Tij[ i][ j] .traf.
Tij[ j][ i] .traf = min_prop * Tij[ j][ i] .traf.
```

14) Impose minimum of 1 erlang between each pair
```
if (Tij[ i][ j] .traf<1.0) Tij[ i][ j] .traf=1.0
if (Tij[ j][ i] .traf<1.0) Tij[ j][ i] .traf=1.0.
```

15) Add new Tij/Tji into sums, so that sums are kept up to date along the way.
```
sumTij[ i] += Tij[ i][ j] .traf
sumTij[ j] += Tij[ j][ i] .traf.
```

16) use handshaking to increase `sumTij`.

17) Test Convergence
If `sumTij` is off by more than 10% for any switch, don't converge; otherwise, set flag to converge. Print progress for each iteration as the total amount of traffic difference between the `sumTij`'s and the `offd_traf`'s.

18) Set up for Step III.
Calculate Egress Traffic and print intermediate results.

STEP III

1) Perform `offd_traf<==>egr_traf` equalization. {Note that this step does not use handshaking, so that decreases/increases in Tij are not mirrored for Tji direction. This is why this step is done last and only once}.

2) Increase or decrease allocation from i to j, depending on j's balance of `offd/egr` traffic. Adjust `Tij[ i][ j] .traf` proportionately to `offd/egr` ratio.

3) Impose minimum of 1 erlang between each pair

4) Renormalize Tij so that it sums to `offd_traf`

5) Impose minimum of 1 erlang between each pair

6) Resum `sumTij` from Tij now that it's normalized.

7) Calculate Egress Traffic and print final results

8) Print Output `TrafDist` File

Inputs none

Outputs please see description of output file "`spr2lecam file`" under the module description section.

Purpose to format and print data for input to lecam.c

Called By `main()`

Calls To none

Local Variables

integer `count, k, l, m, n` loop count variables

Global Variables This routine define the community of interest weighting that describes how much or how little traffic a switch will send to another switch due to their proximity.

Algorithmic Description This function writes out all output variables to the output file " `spr2lecam.step1`" for a list of these variables and the file format see the description of the output file under the module description section.

3.4 tami: Master Loop for the TAMI model

Purpose This is the master loop for the TAMI model. It forms the Monte Carlo loop around the data preparation and LECAM/QTCM modules. It calls each of the modules, passing along the appropriate command line parameters. The keyfile (-k) contains all of the necessary data files to run TAMI. Results of each Monte Carlo bin are stored in TAMI.RESULTS.binX, where X is replaced with the current bin number. Bins may either be Monte Carlo or use a single damage vector. Bin sizes are defined in the keyfile.

Call Syntax `tami -k <key file>` [*options*]

mandatory:	*special syntax:*	*function:*
`-k`	`<key file>`	reads in input file `<key file>`

options:	*special syntax:*	*function:*
`-c`		indicates use of CSI file.
`-d`		turns dynamic overload control on or off
`-m`	`<#>`	offered traffic multiplier
`-e`	`<1> or <2>`	invokes early-post-disaster rules (early1 or early2 only)
`-a`		disables alternate routing scheme for AT&T
`-v`		invokes `INITPREV` in `LECAM` which initiates `LECAM` with previous values
`-n`		invokes the use of the NS/EP matrix
`-r`		indicates regional focused overload
`-x`		cancels print results in `LECAM`
`-l`		turns on LEC TCR in selected LATAs
`-?`		user help—prints call syntax and exits without running

example `tami -k dplive.key -m 10 -e 1 -v` (spaces optional)

Input Files

key file — This file contains hardcoded instructions for opening up and reading in the various data files to be used in this module.

format

comment section:
<comment line>

iec toggle section:
<iec toggles>
`ATT` and/or `MCI` and/or `SPR`

iec bin section:
<opional bin off toggle>, <Bin#>, <# of bin iterations>,
<descriptive text>
`X` or `x Bin1 1 Live PSN`

lec toggle section:
<lec>
`LEC`

lec file section:
<EO damage file>
`eo_damage_file`
<TDM damage file>
`tdm_damage_file`
<EO to AT damage file>
`eo_at_damage_file`
<NSEP toggle>, <NSEP file>
`+N nseptraf.live`
<regional overload toggle>, < regional overload file>
`+R fo_sf_15_10.txt`
<LEC TCR toggle>, < LEC TCR file>
`+L tcr_latas.25`

iec file section: (3 separate sections for each iec)
<IEC indicator>
`ATT` or `MCI` or `SPR`
<at pop size file>
`iec2.lecam.step1.atpopsize`
<SW damage file>
`sw_damage_iec`
<span file>
`spanfile`
<POP damage file>
`pop_damage_iec`
<CSI file>
`csifile` **(ATT only)**
<EO-POP damage file>
`eo_pop_damage_iec`
<AT-POP damage file>
`at_pop_damage_iec`
<POP SW damage file>
`popsw_damage_iec`
<SW homing file>
`sw_hmg.mci` **(MCI only)**
<Qlink file>
`qlink.iec.sf.replaced`
<Traffic Distribution file>
`trafdist94.iec` **(MCI and Sprint only)**

Output Files

TAMI.RESULTS.binX file

This file contains the results of each Monte Carlo bin, for X = bin number

format **see output file description under `keepstats.c`**

Includes

`<stdio.h>`	Standard 'c' defined I/O functions
`<math.h>`	Standard 'c' defined math functions
`"fileio.c"`	User-defined I/O functions; see Appendix A

Constants

`MAXBIN`	10	maximum number of iterations
`MAX_IEC`	3	defined number of interexchange carriers

Global Variables

integer	KEYTOG	indicates keyfile will be input from command line (prompted otherwise)
	DOC	toggle for dynamic overload control (0=off, 1=on)
	EARLY	indicates time period after damage (0=late, 1=early1, 2=early2)
	ALTOFF	toggle for AT&T RTNR (0=on, 1=off)
	CSI	toggle for alternate priority traffic routing for NSEP users using *Commercial Satellite Inter-connectivity(CSI)* (0=off, 1=on)
	INITPREV	toggle to indicate use of variables from previous iteration (0=off, 1=on)
	NSEPMAT	toggle to indicate use of NSEP traffic matrix (0=off, 1=on)
	CANCELPR	toggle to control printing of LECAM results (0=off, 1=on)
	LECTCR	toggle to indicate use of LEC TCR in selected LATAs (0=off, 1=on)
	RTOG	toggle to indicate application of regional focused overload (0=off, 1=on)
	NumBins	number of damage bins
	att	IEC toggle
	mci	IEC toggle
	spr	IEC toggle
character	DPOptions[30]	holds the toggles from the data preparation section
	LecamOptions[500]	holds the toggles from the LECAM section
	SwitchDmgFile[MAX_IEC][80]	holds the name of the IEC switch damage files
	QlinkFile[MAX_IEC][80]	holds the name of the IEC qlink files
	SwHmgFile[80]	holds the name of the MCI Switch Homing file
	CSIfile[80]	holds the name of the CSI Damage file
	regovl_file[80]	holds the name of the Regional Overload file
	lectcr_file[80]	holds the name of the LECTCR file
	nsep_file[80]	holds the name of the LECTCR file
structure	BinList[MAXBIN]	of type *with fields:*
	character	BinList[MAXBIN] .name[50]
	integer	BinList[MAXBIN] .ONOFF
		BinList[MAXBIN] .maxiter
real	TMULT	Offered traffic multiplier - default is 1x

Local Variables

Variables local to `main()`

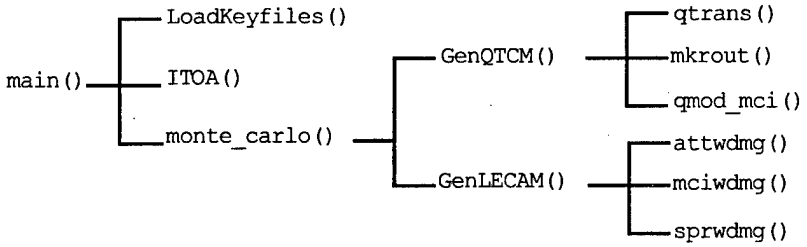
extern	character *optarg	a string containing a single command line argument.
integer	optind	the number of single command line arguments to be processed, supplied externally by the operating system
character	ch	a command line option character
	line[200]	buffer for parsing a single input line from a file
	temp[20]	used to parse a line
	TMULTSTR[10]	holds offered traffic multiplier for character to integer conversion
	keyfile[80]	holds the name of the input key file
	**argv[ ]	an array containing all of the command line arguments

integer	`err=0`	error flag indicating a problem with the command line arguments
	`argc`	the number of command line arguments
	`i, j`	loop counter variables
	`current_bin`	
	`vector_offset`	
	`OPTOG`	indicates command line options
FILE	`*fptr`	file pointer

Component Functions

void	`LoadKeyfile()`	loads input file names from the `<keyfile>` specified in the command line
	`ITOA()`	converts from integer to a character variable
	`GenQTCM()`	runs the QTCM filters to generate the QTCM input files
	`GenLECAM()`	runs the data preparation filters to generate the LECAM input files
	`monte_carlo()`	runs an individual bin loop

Function Tree



Algorithmic Description

This is the master loop for the TAMI model. It forms the Monte Carlo loop around the data preparation and LECAM/QTCM modules. It calls each of the modules, passing along the appropriate command line parameters. The keyfile (-k) contains all of the necessary data files to run TAMI. Results of each Monte Carlo bin are stored in TAMI.RESULTS.binX, where X is replaced with the current bin number. Bins may either be Monte Carlo or use a single damage vector. Bin sizes are defined in the keyfile.

After the command line toggles have been interpreted, the `main()` routine begins by passing the `<keyfile>` argument into `LoadKeyfile()`, which reads in the number and names of the IEC networks being used and the identity of the bins. Next it checks for the following options to see if they have been set 'on/off for logical variables or given a value for integer variables: DOC, TMULT, EARLY, CSI, ALTOFF, NSEPMAT, RTOG, LECTCR, INITPREV, CANCLEPR. It then begins looping through all of the various bins as specified in `NumBins`. Each bin is then passed as an argument to `monte_carlo()`. `Monte_carlo()` runs an individual bin loop using `GenQTCM()` and `GenLECAM()`. The calls `Init()`, which initializes all network variables. `Readfile()` is called next to read in all network data. `Main()` then opens the `<outfile>` and proceeds to perform the core of the algorithmic work in this module.

Inputs

character `keyfile[ ]`

Outputs

returns no formal returns

global	character	`SwitchDmgFile[ ]`	the name of the IEC switch damage files
		`QlinkFile[ ][ ]`	the name of the IEC qlink files
		`SwHmgFile[ ]`	the name of the MCI Switch Homing file
		`CSIfile[ ]`	the name of the CSI Damage file

Purpose This routine loads file names from the keyfile specified in the command line of `tami.c`.

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	loop count variables
	`pos`	indicates line position in the output character variables
character	`line[ 100]`	used for parsing the key file
	`temp[ 10]`	used for parsing the key file
	`temp2[ 100]`	used for parsing the key file
file	`*fptr`	points to an open file

Global Variables

character	`att`	IEC toggle
	`mci`	IEC toggle
	`spr`	IEC toggle

Algorithmic Description

1) Open the file `keyfile` as read only and assign it to `*fptr`.

2) Skip the comment lines in the file.

3) Find the IEC section of the file, determine which IECs are being used, and set toggles on.

4) Load damage bins.
   For each line in the keyfile which contains a damage bin read the line and determine the number of bin iterations and the names of the bins.
   For each line in the keyfile which contains LEC infomation check for:
   a regional overload file.
   a LECTCR file.
   a NSEP file.

5) Load IEC carrier file names to specific line positions in the following files for each carrier:

AT&T:	`SwitchDmgFile, CSIfile, QlinkFile;`
MCI:	`SwitchDmgFile, SWHmgFile, QlinkFile;`
Sprint:	`SwitchDmgFile, QlinkFile`

Inputs

integer	c	any integer value up to 4 digits in length

Outputs

returns	character	character representation of an integer

Purpose This routine converts an integer to a string. It accepts any integer up to 9999, converts each digit to a character, and returns a pointer to the resulting string

Called By
```
main()
monte_carlo()
GenQTCM()
GenLECAM()
```

Calls To none

Local Variables

character	s[10]	holds the return value

Global Variables none

Algorithmic Description This function returns the character representation of an integer input by using the standard c utility function `sprintf()`.

Inputs

integer	bin	damage bin
	vector_offset	used to parse a line of input
	maxiter	maximum number of damage vectors for samping
character	name[ ]	holds name of output file
	keyfile[ ]	holds name of input file

Outputs

Purpose

Called By main()

Calls To
GenQTCM()
GenLECAM()
ITOA()
system()

Local Variables

integer	count	loop count variable
	mcbin	monte carlo bin
	target	target number of iterations
	a	used to parse a line of input
	z	used to parse a line of input
	c1	counter variable
	c2	counter variable
real	error	error level
char	temp[200]	used to hold a line of input
	line[100]	used to hold a line of input
	temp2[100]	used to hold a line of input
	ch	used as a flag for the input file data
FILE	*fptr	points to an open file
	*csiptr	points to a csi file
	*outptr	points to an output file

Global Variables none

Algorithmic Description

This routine runs an individual bin loop. This bin may either be a Monte Carlo bin requiring multiple LECAM runs, or a sinle damage bin requiring only one LECAM run. Results of each bin are written to TAMI.RESULTS.bin#, where # is the number of the specific bin being run.

1) Initialize loop variables. Begin by sampling 3 Monte Carlo runs and adding more as necessary. If the maximum number of damage vectors in this bin equals 1, then

it is assumed to be a single static damage vector (i.e., non-Monte Corlo) and is calculated by itself. Set the Monte Carlo flag (mc) accordingly: 1=monte carlo, 0=static.

2) Store each iterations TAMI.lecamlog file in a running file. At the end, copy this running file and the final TAMI.tally file into a file specifically for this iteration.

3) Loop while haven't done enough runs and still have more vectors to sample.

4) Call `GenQTCM()`

5) Call `GenLECAM()`

6) Cut current CSI damage vector for LECAM

7) Merge the data preparation files using a command line system call.

8) Run LECAM using a command line system call.

9) Keep stats on all blockages calculated, but test convergenceon Bpots. The keepstats module will tabulate all BIGB* variables and will echo the error of BIGBPOTS to the screen for convergence testing below.

10) Add the TAMI.lecamlog file to TAMI.runstore with a header.

11) End of the Monte Carlo sampling loop.

12) Store the TAMI.runstore and final TAMI.tally in a file for this bin.

13) Print results.

Inputs

integer	`vector`	holds a single damage vector
character	`keyfile`	input data file

Outputs **see input files for QTCM in the QTCM Programmers Manual; Reference 2**

Purpose This routine runs the QTCM filters to generate the QTCM input files.

Called By `monte_carlo()`

Calls To
`ITOA()`
`system()`

Local Variables

integer	`pos`	used to parse the input file
character	`temp[ 100]`	used to parse the input file
FILE	`*fptr`	points to the input file

Global Variables none

Algorithmic Description This routine runs the QTCM filters to generate the QTCM input files.

1) Run all 3 qtrans. These files will have Section 7
2) start in matrices at postion 0
3) Run each qtrans specified by the IEC toggles
4) If MCI is being run then also run `mkrout()`
5) If MCI is being run then also run `qmodmci()`
6) Rename `qtrans()` files for AT&T and Sprint
7) Rename `mkrout()` file for MCI

Inputs

integer	`vector`	holds a single damage vector
character	`keyfile`	input data file

Outputs **see input files for lecam.c, Section 3.11**

Purpose This routine runs the data preparation filters to generate the LECAM files

Called By `monte_carlo()`

Calls To
`ITOA()`
`system()`

Local Variables

character	`temp[ 100]`	used to piece together a command line for running a system command
	`TAIL[ 150]`	used to parse data from the keyfile

Global Variables none

Algorithmic Description This routine runs the data preparation filters to generate the LECAM files

3.5 mkrout: derive QTCM input file for MCI

Purpose This module determines end-to-end connectivity through MCI's hierarchical backbone.

Call Syntax `mkrout -k <key file>  [ options]`

mandatory:	*special syntax:*	*function:*
`-f`	`<file list>`	reads in input files `<file list>`
`-d`	`<1>, <2>, ..., <99>, <100>`	reads in damage iteration. Must be between 1 and 100 inclusive.

options:	*special syntax:*	*function:*
`-?`		user help—prints call syntax and exits without running

example `mkrout -f sw_fl lk_fl q_fl out_fl -d 50` (spaces optional)

Input Files The files listed below are input through a command line option, formats and explanation can be found in the **QTCM Programmers Manual; Reference 2**

`mainfile`	file holding list of input file names
`swfile`	switch input file
`linkfile`	linking input file
`qtcmfile`	qtcm input file

Output Files

`outfile` This file contains the results of the MCI QTCM template, with hierarchial routing, output file

format **see QTCM Programmers Manual; Reference 2**

example

Includes

`<stdio.h>`	Standard 'c' defined I/O functions
`<stdib.h>`	Standard 'c' defined functions
`<string.h>`	Standard 'c' defined string functions
`"fileio.c"`	User-defined I/O functions

Constants

`SWITCH_MAX`	150	maximum number of switches
`CLLI_LNG`	5	maximum clii code length
`CL_13_MAX`	1000	maximum number of switch pairs between class1 and class3 switches
`CL_33_MAX`	8000	maximum number of switch pairs between class3 and class3 switches
`CL_11_MAX`	100	maximum number of switch pairs between class1 and class1 switches
`LINE_LENGTH`	200	maximum line length
`SW_REC_LEN`	9	switch record length

Global Variables

integer	`dmg_iter`		number of iterations
	`num_sw`		number of switches
	`num_11=0`		number of class1 to class1 switch pairs
	`num_13=0`		number of class1 to class3 switch pairs
	`num_33=0`		number of class3 to class3 switch pairs
	`nroute[ SWITCH_MAX][ SWITCH_MAX][ 8][ 2]`		number of possible routes between 2 switches
character	`line[ LINE_LENGTH]`		length of input line
structure	`sw` of type `sw_struct`		
	with fields:		
	integer	`class`	switch class
		`cl1_idx`	index of the class1 switch, from 1,n
	character	`clli`	switch clli code
		`class1`	switch class
structure	`link_struct` of type `link_struct`		
	with fields:		
	integer	`sw1`	originating switch
		`sw2`	terminating switch
link_struct	`c11[ CL_11_MAX]`		vector of class1 to class1 switch pairs
	`c13[ CL_13_MAX]`		vector of class1 to class3 switch pairs
	`c33[ CL_33_MAX]`		vector of class3 to class3 switch pairs
FILE	`*mainfile`		points to the input file containg the names of the files needed in this routine
	`*swfile`		points to the name of the switch file
	`*linkfile`		points to the name of the link file
	`*qtcmfile`		points to the name of the qtcm data file
	`*outfile`		points to the name of the output routing file

Local Variables

Variables local to `main()`

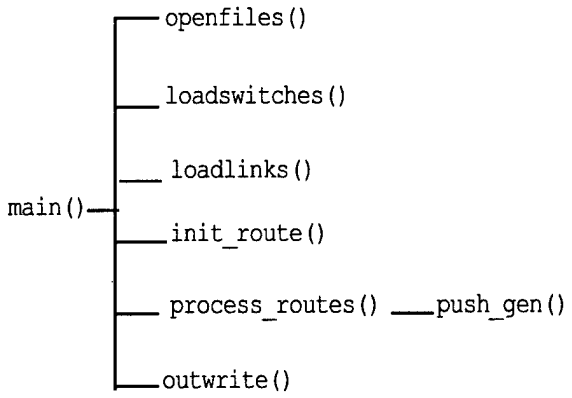
extern	character	`*optarg`	a string containing a single command line argument.
	integer	`optind`	the number of single command line arguments to be processed, supplied externally by the operating system
character	`ch`		a command line option character
	`filelist[ 80]`		holds the name of the input key file
	`**argv[ ]`		an array containing all of the command line arguments
integer	`tog_err=0`		error flag indicating a problem with the command line arguments
	`tog_filelist=0`		error flag indicating a problem with the command line filelist argument

`tog_dmg_iter=0`	error flag indicating a problem with the command line damage iteration argument
`argc`	the number of command line arguments

Component Functions

`openfiles()`	loads input file names from the `<filelist>` specified in the command line
`loadswitches()`	loads input switch data
`loadlinks()`	loads input link data
`initroute()`	initializes traffic routes
`process_routes()`	processes traffic routes
`push_gen()`	determines alternate traffic routes
`push_13()`	not used

Function Tree



Algorithmic Description

This module is used solely to complete the network representation for the hierarchial MCI network. After declaring functions, global and local variables the following steps are executed:

1) Read in and check validity of command line arguments.

2) Open switch and link input files and routing table output file, call `openfiles()`

3) Load switch records, call `loadswitches()`.

4) Load link records, call `loadswitches()`.

5) Initialize routing table, call `initroute()`.

6) Process routing table, call `process_routes()`.

7) Output routing table, call `outwrite()`.

8) Close all files.

Inputs

character	*files	pointer to a file list

Outputs

returns no formal returns

FILE	*mainfile	points to the input file containg the names of the files needed in this routine
	*swfile	points to the name of the switch file
	*linkfile	points to the name of the link file
	*qtcmfile	points to the name of the qtcm data file
	*outfile	points to the name of the output routing file

Purpose This routine loads file names from the input file specified in the command line of mkrout.c.

Called By main()

Calls To none

Local Variables

character	tempfile[80]	used for parsing the input file

Global Variables none

Algorithmic Description

1) Open the input file files as read only and assign it to mainfile.
2) Scan the first line of mainfile for the name of the switch file, open it as read only and assign it to swfile.
3) Scan the next line of mainfile for the name of the link file, open it as read only and assign it to linkfile.
4) Scan the next line of mainfile for the name of the qtcm file, open it as read only and assign it to qtcmfile.
5) Scan the next line of mainfile for the name of the output file, open it as write only and assign it to outfile.

Inputs none

Outputs

structure `sw` of type `sw_struct`
with fields:

integer	`class`	switch class
	`cl1_idx`	index of the class1 switch, from 1,n
character	`clli`	switch clli code
	`class1`	switch class

returns no formal returns

Purpose This routine loads the class and index value for each switch pair

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	loop count variables
	`len`	lenght of a line of input from the switch file

Global Variables

character	`line[ LINE_LENGTH]`	length of input line

Global Constants

`SW_REC_LEN`	length of a switch record

Algorithmic Description

1) Initialize first record from the `swfile`
2) Process the switch clli codes and switch class, record by record
3) Keep track of the number of switches.

Inputs none

Outputs

structure `sw` of type `sw_struct`
with fields:

integer	`class`	switch class
	`cl1_idx`	index of the class1 switch, from 1,n
character	`clli`	switch clli code
	`class1`	switch class

structure `link_struct` of type `link_struct`
with fields:

integer	`sw1`	originating switch
	`sw2`	terminating switch

link_struct	`c11[ CL_11_MAX]`	vector of class1 to class1 switch pairs
	`c13[ CL_13_MAX]`	vector of class1 to class3 switch pairs
	`c33[ CL_33_MAX]`	vector of class3 to class3 switch pairs

returns no formal returns

Purpose to parse each record of the input file into 2 index values

Called By main()

Calls To none

Local Variables

integer	`i`	loop count variable
	`idx1`	first switch index
	`idx2`	second switch index
	`dmg`	switch damage value, 0 or 1
	`num_alive`	number of surviving switches
	`j`	loop count variable
	`found`	used to determine uniqueness of swith pairs

Global Variables

integer	`num_11=0`	number of class1 to class1 switch pairs
	`num_13=0`	number of class1 to class3 switch pairs
	`num_33=0`	number of class3 to class3 switch pairs
character	`line[ LINE_LENGTH]`	length of input line

Algorithmic Description

1) Parse each record of the input link file into 2 index values for each link between switch pairs and the damage value associated with that pair, 0 or 1.

2) If no damage, then add the switch index to the appropriate case:
   - class1--> class1
   - class1--> class3
   - class3--> class3

3) In each case only add unique values to swithc index lists

4) Print out switch statistics.

Inputs none

Outputs

integer	`nroute[ ][ ][ ][ ]`	number of possible routes between 2 switches

returns no formal returns

Purpose to initialize all traffic routes in mci

Called By `main()`

Calls To none

Local Variables

integer	`i, j, k`	loop count variables

Global Variables none

Global Constants

`SWITCH_MAX`	`150`	maximum number of switches

Algorithmic Description This function initializes all values of `nroute` for each switch pair with two values: 99 and 0.

Inputs none

Outputs

structure `sw` of type `sw_struct`
with fields:

integer	`class`	switch class
	`cl1_idx`	index of the class1 switch, from 1,n
character	`clli`	switch clli code
	`class1`	switch class

structure `link_struct` of type `link_struct`
with fields:

integer	`sw1`	originating switch
	`sw2`	terminating switch

link_struct	`c11[ CL_11_MAX]`	vector of class1 to class1 switch pairs
	`c13[ CL_13_MAX]`	vector of class1 to class3 switch pairs
	`c33[ CL_33_MAX]`	vector of class3 to class3 switch pairs

returns no formal returns

Purpose to process the initialized traffic routes in mci

Called By `main()`

Calls To `push_gen()`

Local Variables

integer	`i, j, k`	loop count variables

Global Variables

integer	`num_11=0`	number of class1 to class1 switch pairs
	`num_13=0`	number of class1 to class3 switch pairs
	`num_33=0`	number of class3 to class3 switch pairs

Algorithmic Description This function process the traffic routes for each switch pair based on the classes of the two switch pairs:

- class1-->class1
- class1-->class3
- class3-->class3

1) Process class3-->class3 direct routes

2) Process class3-->class3 via routes both homed to the same class1

3) If two class3 switches share same class1, and the class1 is the home class1 for one of them, then process valid class3--> class3 via route

4) Process class3-->class3 via routes each homed to a unique class1

5) If two different class3 switches have links to two different class1 switches, and if the class1 switches are the home switches of their respective class3 switches, then process valid class3--> class3 route. In other words, there is a class3-->home class1-->home class1'-->class3' route

6) Check for existence of class1-->class1' link.

7) Process 2 cases where POPs home directly to a class1 switch. In these cases, a class1 must be able to route to class 3 and other class 1 switches.

8) Process class1-->class1direct routes

9) Process class1-->class1-->class3 direct routes

10) If a class3 goes to its home class 1, and this class 1 is connected to another class 1, then process valid class1-->class1'-->class3' route

11) Class1 given by c11[k].sw1 is the home class1for class3 given by `c13[ i] .sw2.`

Inputs

integer	A	originating switch
	Z	terminating switch
	via	via switch
	hops	number of switches in between

Outputs

integer	nroute[][][][]	number of possible routes between 2 switches

returns no formal returns

Purpose This function exchanges an alternate route for a more direct route

Called By process_routes()

Calls To none

Local Variables

integer	done	logical completion flag
	i	loop count variable

Global Variables none

Algorithmic Description This function exchanges an alternate route for a more direct route.

Inputs

FILE	`*qtcmfile`	points to the name of the qtcm data file

no formal inputs

Outputs

FILE	`*outfile`	points to the name of the output routing file

returns no formal returns

Purpose This function writes the qtcmfile into the outfile

Called By `main()`

Calls To none

Local Variables

integer	`m1`	loop counter variable
	`m2`	loop counter variable
	`m3`	loop counter variable
	`sect3_flag=0`	output file flag
	`sect4_flag=0`	output file flag

Global Variables

character	`line[ LINE_LENGTH]`	length of input line

Algorithmic Description This function writes the qtcmfile into outfile until SECTION 03 of the qtcm file is reached. It then proceeds to position the qtcmfile at SECTION 04, and then writes the qtcmfile into outfile from SECTION 04 to EOF.

3.6 qtrans_gen: Generate QTCM Files

Purpose To create a qtcm input file from the qswitch and qlink file format used for the ncam to qtcm conversion process

Call Syntax `qtrans_gen.f`

example `qtrans_gen.f`

Input Files Descriptions of the input file can be found in **QTCM Programmer's Manual; Reference 2.**

`qlink.iec`
`switch_iec.data.dmg`

Output Files Descriptions of theoutput file can be found in **QTCM Programmer's Manual; Reference 2.**

`outfile`	This file contains the results of output file

Includes none

Constants none

Global Variables Definitions of the global variables can be found in **QTCM Programmer's Manual; Reference 2.**

real
```
RGOS
DGOS
EGOS
PARC
PROP
WTGSAT
WTGMIL
EMIN
LBF
LVBF
STABLE
TINY
TINYBF
E1FRAC
E2FRAC
OFLOAD
NTRUNK
NOTRNK
```

integer
```
*4 MNDCNC
*4 GROUP
*4 NNOD
ICON1
ICON2
*4 CST
KDIST
*4 IX
```

```
NROUTE
*4 ITRUNK
IOTRNK
*4 SWITCH
*4 NOUSER
NOSWIT
*4 MAXLEN
LTYPE
NOITS
MAXSAT
DSNCNT
*4 FLGPTR(15)
NOSBST(15)
NOFLGS
NOSCTS
*4 FPTR
CHAN(8)
FLSTAT(8)
NODE(8)
FILSCT(15,8)
```

array

```
OFLOAD(250,250)
MNDCNC(250,250,4)
GROUP(250,250)
CST(250,250)
KDIST(250,250)
NROUTE(250,250,8,2)
NTRUNK(250,250,3)
ITRUNK(250,250,3)
IOTRNK(250,250,3)
SWITCH(250,5)
NOTRNK(250,250,3)
NOUSER(250)
NOSWIT(250)
DATAIN(20)
DATAOK(20)
```

logical

```
AMEND
ANLYSE
DATAIN
DATAOK
```

Local Variables

Definitions of the local variables can be found in **QTCM Programmer's Manual; Reference 2.**

character

```
*41 TEXT(10)
COSTXT(2)
SUBTXT(4)
DAMTXT(5)
*30 DESC(2)
*15 SCFTR
*80 OUTFIL
 SWTFIL
 SPNFIL
* 5 VERNUM
*325 NLINE
PLINE
PDMG
```

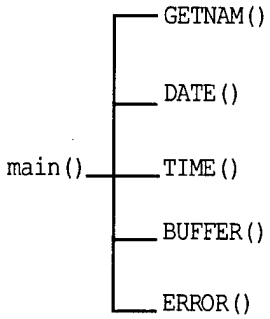
```
          *80 DUMMY1
          *  2 NO
          *  6 ONEWAY

integer   *  4 OUT
          IN1
          IN2
          TOTSCT
          VALSCT
          COUNT
          *  4 DUMMY2
          *  4 ICOUNT
          ISRC
          IDEST
          ICAP
          IDEFCAP

  real    RT1

logical   OK
          NODAM
          ONEW

  data    VERNUM    / ' 4.0 ' /
          NO        / 'NO' /
          ONEWAY    / 'ONEWAY' /
          TEXT(1)   /'SECTION 01 - GENERAL INFORMATION'/
          TEXT(2)   /'SECTION 02 - NETWORK COSTS AND DISTANCES'/
          TEXT(3)   /'SECTION 03 - ROUTING TABLE'/
          TEXT(4)   /'SECTION 04 - MINIMUM TRUNK GROUP'/
          TEXT(5)   /'SECTION 05 - MANDATORY TRUNK CONNECTIONS'/
          TEXT(6)   /'SECTION 06 - TRUNK SIZE'/
          TEXT(7)   /'SECTION 07 - OFFERED LOAD'/
          TEXT(8)   /'SECTION 08 - SWITCH DETAILS'/
          TEXT(9)   /'SECTION 09 - DAMAGE & PREFERENCE ROUTING'/

          COSTXT(1) /'SUBSECTION 1 - COSTS'/
          COSTXT(2) /'SUBSECTION 2 - DISTANCES'/

          SUBTXT(1) /'SUBSECTION 1 - SATELLITE'/
          SUBTXT(2) /'SUBSECTION 2 - TERRESTRIAL'/
          SUBTXT(3) /'SUBSECTION 3 - MILITARY'/
          SUBTXT(4) /'SUBSECTION 4 - OVERALL'/

          DAMTXT(1) /'SUBSECTION 1 - USERS'/
          DAMTXT(2) /'SUBSECTION 2 - SWITCHES'/
          DAMTXT(3) /'SUBSECTION 3 - TRUNKS - SATELLITE'/
          DAMTXT(4) /'SUBSECTION 4 - TRUNKS - TERRESTRIAL'/
          DAMTXT(5) /'SUBSECTION 5 - TRUNKS - MILITARY'/

          DESC(1) /' DATA HAS BEEN VALIDATED'/
          DESC(2) /' DATA HAS NOT BEEN VALIDATED'/

          SCFTR /'SCALING FACTOR '/
          TOTSCT /19/
```

Component Functions

Definitions of the component functions can be found **QTCM Programmer's Manual; Reference 2.**

```
GETNAM()
DATE()
TIME()
BUFFER()
ERROR()
```

Function Tree



Algorithmic Description

This module is used to produce a qtcm input data file and is coded in FORTRAN

1) Open user-named input switch, span files and output files.
2) Calculate number of iterations in switch file and Cross check with the number of iter's in span file.
3) Give option for directionality (mci=oneway, att & spr=bidirect).
4) Give option to ignore damage (sets flag so that section 9 is given default values from init).
5) Read in desired iteration from user.
6) Calculate the number of records in the switch file. This number is the number of nodes (nnod).
7) Prompt for an average link size to be used as a default if 9999 is read in for link size.
8) Call init.
9) Output header.
10) Output general data.
11) Calculate qtcm data from ncam data.
12) Calculate damaged switches—section 9.2.

13) Calculate trunk capacity—section 6.2.

14) Test for damage to link or either endpoint.

15) Set damaged trunk capacity—section 9.4.

16) Derive section 4, 5.2, 5.4 and 7 from section 6.2 note: section 7 is derived to have a "trickle current". This is used to get the ppmeanfile for mci as input into "mciwdmg".

17) Output network cost and distance matrices.

18) Write routing table.

19) Write minimum group size.

20) Output mandatory connectivity arrays.

21) Output trunk size arrays.

22) Write offered load.

23) Scale offered load if necessary.

24) Write switch details.

25) Output damage and preferential routing data.

26) Write out completion message.

27) Format statements

Definitions of the component functions can be found in **QTCM Programmer's Manual; Reference 2.**

Definitions of the component functions can be found in **QTCM Programmer's Manual; Reference 2.**

Definitions of the component functions can be found in **QTCM Programmer's Manual; Reference 2.**

Definitions of the component functions can be found in **QTCM Programmer's Manual; Reference 2.**

Definitions of the component functions can be found in **QTCM Programmer's Manual; Reference 2.**

3.7 attwdmg: Pre-process Damaged AT&T Network and Traffic

Purpose This module performs the data preparation calculations required to assess the affects of damage to the LEC homing chains and end-to-end traffic matrix for the AT&T network. Many of the engineered traffic variables computed by `attlive` are adjusted to reflect post-damage engineered traffic under Early and Late scenarios. For a regional analysis, this module computes the focused overload offered load matrix through AT&T. The output file from this module is merged with the other IEC files in the `merge` module before being input into the `lecam` module.

Call Syntax `attwdmg -k <key file> [ options]`

mandatory:

`-d`	use damage vector `<number>` for this run
`-k`	reads in input file `<key file>`

options:

`-b`	blast-type damage: use physical diversity model for HU damage
`-e`	specify early scenario <1> or <2>
`-i`	no longer used
`-r`	reads in regional overload input file `<reg file>`
`-s`	specify non-default random number generator stream using `<stream>` from 1 to 15
`-?`	user help—prints call syntax and exits without running

example `attwdmg -k tamirun.key -d5`

Input Files The list of input files required by attwdmg is supplied in the keyfile. See Appendix B (TAMI keyfile description) for these files.

reg file In addition to the files in the keyfile, for a regional overload run, the -r option will specify a regional overload file, <reg file>. The first line of this file specifies the default (Region 0) overload. Unless specified in later regions, all LATAs will be overloaded based on this default. The second line defines Region 1 and its overload value. The next line contains a list of space-separated LATA numbers that are members of Region 1. Subsequent regions are defined similarly, up to a maximum of five regions.

format

R0, <region overload>	default region line
<Region number>, <region overload>	subsequent region line (c2, 1x, f)
<LATA number>, . . . , <LATA number> (i3, 1x, . . ., i3, 1x, i3)	with associated LATAs

example

`R0 1.25`	defines 1.25 overload for default LATAs
`R1 5.0`	defines 5.0 overload for region 1 LATAs
`132 133 134`	region 1 LATAs defined here as a list
`R2 10.0`	defines 10.0 overload for region 2 LATAs
`251 291 237 252`	region 2 LATAs defined here as a list

Output Files att2lecam.out—multisection output file described in function Outwrite

Includes

"fileio.c"	includes input/output utility functions—see Appendix A
"lecam.h"	defines lecam constants and variables—see Section 3.11
"waglib.h"	defines random number generator functions, constants, and data structures—not used

Constants

MAX_BIN 3	Not used
NUM_TEMP 25	Number of sections in output file
TEST_SW #	Number of a switch for which debug statements will be printed

constants defined in lecam.h (network sizing variables) and used in attwdmg:

TRUE	1	
FALSE	0	
MAX_AT	1000	Maximum number of ATs in network
MAX_EO	19250	Maximum number of EOs in network
ATPOPMAX	9	Maximum number of POPs a single AT may home to
MAX_POP	575	Maximum number of POPs in network
MAX_SWITCH	130	Maximum number of IEC switches in network
MAX_LATA	1000	Maximum number of LATAs
MAX_REGION	5	Maximum number of regions defined for regional overload

Global Variables

FILE	*inptr	File pointer for an input file
	*outptr	File pointer for the output file
	*fptr	File pointer for any file
integer	KEYTOG	toggle for -k option
	rtog	toggle for -r option
	NUMEO	number of EOs in network
	NUMAT	number of ATs in network
	NUMPOP	number of POPs in network
	NUMSW	number of backbone switches in network
	NUMREGIONS	number of regions defined for regional overload
	ATPOPSIZE	maximum number of AT-POP homing segments
	HAP[MAX_AT] [ATPOPMAX]	AT-POP homings
	AT[MAX_EO]	homed AT for an EO
	ATPOP[MAX_EO]	homed POP through the AT for an EO
	HUPOP[MAX_EO]	homed high-usage POP for an EO
	SW[MAX_EO]	homed IEC switch for an EO
	TYPE[MAX_EO]	homing type for an EO
	LATA[MAX_EO]	LATA designation for an EO
	popswitch[MAX_POP]	homed IEC switch for POP
	IMT[MAX_SWITCH] [MAX_SWITCH]	matrix of IMT TGs between IEC switches
	IMT2[MAX_SWITCH] [MAX_SWITCH]	copy of IMT matrix used for data checking purposes
	IMTdam[MAX_SWITCH] [MAX_SWITCH]	damaged IMT matrix
	SUM_IMT[MAX_SWITCH]	total IMT TGs originating or terminating at each IEC switch
	SAP[MAX_AT] [ATPOPMAX]	size of AT-POP TG for each POP homed to AT
	SEP[MAX_EO]	size of the direct EO-POP TG, if exists

	PhysDiv[MAX_EO]	0/1 indication of whether the HU TG, if existing, is physically diverse from the AT
	EOalive[MAX_EO]	0/1 indication of survival of EO
	ATalive[MAX_AT]	0/1 indication of survival of AT
	POPalive[MAX_POP]	0/1 indication of survival of POP
	SWalive[MAX_SW]	0/1 indication of survival of IEC switch
	EOATalive[MAX_EO]	0/1 indication of survival of EO-AT span
	ATPOPalive[MAX_AT] [ATPOPMAX]	0/1 indication of survival of AT-POP span
	EOPOPalive[MAX_EO]	0/1 indication of survival of EO-POP span
	POPSWalive[MAX_POP]	0/1 indication of survival of POP-SW span
	LReg[MAX_LATA]	Indicates which Region each LATA is in
	EOReg[MAX_EO]	Indicates which Region each EO is in
float	TAP[MAX_AT] [ATPOPMAX]	traffic engineered for SAP TGs
	TEP[MAX_EO]	traffic engineered for SEP TGs
	TEA[MAX_EO]	traffic engineered between EO and AT
	ACSlec[MAX_SWITCH]	traffic offered from LEC to each IEC switch
	ACSlec_live[MAX_SWITCH]	traffic offered from LEC to each IEC switch without considering network damage
	ACSlec_dam[MAX_SWITCH]	traffic offered from LEC to each IEC switch including effects of damage
	ACSiec[MAX_SWITCH]	traffic offered to IEC switch for inter-switch transport
	ACStest[MAX_SWITCH]	temporary testing of ACS
	PropIntra[MAX_SWITCH]	proportion of inter-switch traffic at each IEC switch
	D3[MAX_POP]	traffic offered from LEC to each POP
	D3live[MAX_POP]	traffic offered from LEC to each POP in the case of no network damage
	D3test[MAX_POP]	temporary testing of D3
	ACPiec[MAX_POP]	traffic through each POP engineered for inter-switch transport
	AC[MAX_EO]	traffic offered from each EO
	Lmult[MAX_LATA]	indicates regional overload multiplier for each LATA
	Rmult[MAX_REGION]	indicates regional overload multiplier assigned to each region
	ACSRegion[MAX_SWITCH] [MAX_REGION]	indicates traffic offered to each switch from each region
	AvgSwRmult[MAX_SWITCH] [MAX_REGION]	indicates the traffic-weighted average multiplier applied to each element of ACSRegion
	RmultMatrix[MAX_REGION] [MAX_REGION]	indicates the traffic multiplier applied to traffic from one region to another
double	SS[MAX_SWITCH] [MAX_SWITCH]	the main switch-to-switch proportions matrix that describes how traffic is distributed across the IEC backbone
	SSreg[MAX_SWITCH] [MAX_SWITCH]	the SS matrix after being modified to reflect regional focused overload
	SSrow[MAX_SWITCH]	sum of an SS row, used for normalizing the matrix
	F0[MAX_EO] [2]	Engineered access traffic at an EO divided by HU and final engineered proportions
	F0dmg[MAX_EO]	Engineered access traffic at an EO with damage considered
	RegF0dmg[MAX_EO]	Engineered access traffic at an EO with damage and regional overload considered

character	`template[ NUM_TEMP] [ 15]`	holds the labels for each section of the output file
	`in_file[ 80]`	name of network file output by `attlive`
	`qlink_file[ 80]`	name of qlink file
	`eodmg_file[ 80]`	name of EO damage file
	`atdmg_file[ 80]`	name of AT damage file
	`popdmg_file[ 80]`	name of POP damage file
	`swdmg_file[ 80]`	name of IEC switch damage file
	`eoatdmg_file[ 80]`	name of EO-AT span damage file
	`eopopdmg_file[ 80]`	name of EO-POP high usage span damage file
	`atpopdmg_file[ 80]`	name of AT-POP span damage file
	`popswdmg_file[ 80]`	name of POP-switch span damage file
	`regovl_file[ 80]`	name of regional overload file, if used

Local Variables

Variables local to `main()`:

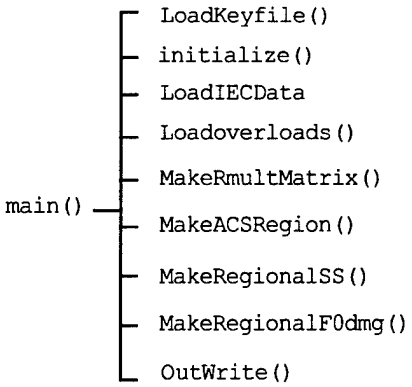
extern	integer `optind, argc`	command line options counters
	character `*optarg, ** argv[ ]`	reference to command line option string
character	`c`	general single character variable
	`out_file[ 80]`	name of output file
	`keyfile[ 80]`	name of keyfile
	`line[ 200]`	buffer for line of input from file
	`temp[ 5]`	temporary text buffer
integer	`err`	toggle to indicate error in command line arguments
	`etog`	toggle to indicate -e option
	`stog`	toggle to indicate -s option
	`ttog`	toggle to indicate -t option
	`itog`	toggle to indicate -i option
	`otog`	toggle to indicate -o option
	`qtog`	toggle to indicate -q option
	`dtog`	toggle to indicate -d option
	`btog`	toggle to indicate -b option
	`stream`	random number stream (default = 10)
	`dvector`	damage vector number to use (default = 1)
	`early`	indicates Early 1 or 2 cases (default = 0)
	`done`	used as a boolean in loops
	`num_dead, num_tot`	counts node and span damage summary statistics
	`cap_live, cap_tot, cap`	counts IEC TG damage summary statistics
	`i,j,k,l,pm,p,m,n`	general loop count variables
	`test,surv,dummy`	general temporary variables
float	`sum_traf, sum_reg_traf`	variables to count total traffic in the network

Component Functions

`LoadKeyfile()`	Loads AT&T input file list from Keyfile
`initialize()`	Initializes all network and traffic variables
`LoadIECData()`	Loads input network file (output from attlive)
`Loadoverloads()`	Loads regional overload input file
`MakeRmultMatrix()`	Calculates the RmultMatrix structure
`MakeACSRegion()`	Calculates the ACSRegion value

`MakeRegionalSS()`	Calculates the backbone distribution matrix for regional overload
`MakeRegionalF0dmg()`	Calculates the EO access traffic for regional overload
`OutWrite()`	Writes network and traffic variables to output file

Function Tree



Algorithmic Description

Introduction—inputs and options: Module `attwdmg` is used to apply a given damage scenario to the undamaged AT&T network description and traffic matrix. It takes as input the undamaged AT&T network description and traffic matrix calculated by `attlive`. In addition, it requires a number of input damage files for each type of node and link (e.g., EO damage file, AT damage file, IEC switch damage file, etc.). Each damage file contains multiple damage scenarios, or vectors. The `tami` module passes the damage vector to be used when it invokes the `attwdmg` module in the -d option. Refer to Section 2 to identify the context of `attwdmg` within the overall TAMI data flow.

`Attwdmg` offers the choice of two sets of assumptions to create a post-damage traffic matrix, as specified by the `-e` option. The "early" assumption assumes that calls are attempted to both connected and unconnected destinations. The "late" assumption assumes that sufficient time has elapsed for users to learn which destinations are unreachable and to stop attempting to call these destinations; calls are only placed to connected destinations.

In addition to assessing the AT&T network and traffic matrix for damage, `attwdmg` can be used to create a regional focused overload traffic matrix if the `-r` option supplies a regional overload file. This input file is used to define a region as a set of LATAs and to describe the overload multiplier that should be used for traffic within the region, traffic leaving the region, and traffic entering the region. Up to five regions may be defined.

Output—`Attwdmg` creates a single, multi-section output file, usually called att2lecam.out, describing the damaged AT&T network, the post-damage traffic matrix, and a focused overload traffic matrix. This file is combined by the `merge` module with the output files for other IECs being analyzed before being passed to the `lecam` module for analysis.

Algorithm—The operation of `attwdmg` is described by the following five stages:

Stage 1: Process arguments, initialize, and load input files.

1.1 Command line arguments are read in and processed for legal combinations

1.2 The keyfile, a mandatory option, is opened and processed with a call to `LoadKeyfile()`

1.3 All global variables are initialized with a call to `initialize()`

1.4 Live network data (the output from `attlive`) is read in using a call to `LoadIECData()`.

1.5 If a regional overload is being performed (`-r` option), then the regional overload file is read in. This file describes the LATAs that comprise a region and the overloads that should be used into, out of, and within the region. `MakeRmultMatrix()` is called to make a region-to-region traffic multiplier matrix.

1.6 Damage vectors are read in one type at a time for all node and (optional) span damage types. Note that the damage vectors in the qlink file reflect pre-assessed damage results for switch-to-switch transmission paths in the IEC backbone. Therefore, individual backbone spans do not need to be considered in `attwdmg`

Stage 2: Assess affect of damage in access region

2.1 The first step is to damage EO-POP spans if they are not physically diverse from the AT, and the path through the AT is damaged. This only applies to type 3 EOs.

2.2 Next, damage is assessed by following homing chains up from an EO depending on the EO type:

2.2.1 For type 1 EOs (HU-only homing), if any link along the HU homing fails, then traffic (AC[k]) cannot make it up that link from the EO. All affected variables are updated to reflect this. In the Early case, EO access traffic is only set to 0 (AC=0) if the EO itself is damaged. If not, the EO will offer traffic, but it will not be able to make it up the homing chain. In the Late case, an EO will not offer traffic if it is disconnected from the network in any way.

2.2.2 For type 2 EOs (AT-only homing), if any link along the AT homing fails, then traffic will not make it up that homing from the EO. All affected variables are updated to reflect this depending on where the damage occured along the homing. For the Early case, AC=0 only if the EO itself is damaged. For the Late case, AC=0 if there is any damage along the homing chain.

2.2.3 For type 3 EOs (HU homing with overflow through an AT homing), both homings must fail to block EO access traffic from arriving at the POP and switch. If only the final homing fails, then all traffic is offered along the HU homing. If only the HU homing fails, then all traffic is offered along the final (AT) homing. In all cases, damage along either or both homing chains is reflected by updating appropriate trunk and traffic variables. As before, the distinction is drawn between the Early and Late cases.

Stage 3: Propogate affect of damage to egress region

In stage 2, access traffic was reduced at various points between the access EO and the access IEC switch due to damage in the access region. Stage 3 propogates this access traffic through the IEC backbone, building the switch-to-switch proportions matrix. An important observation for the Late case is that calls are not attempted to unconnected destinations. Therefore access traffic at each switch must be further reduced by the traffic-weighted proportion of destinations that are unreachable.

3.1 The switch-to-switch backbone distribution matrix, SS, is calculated in this step for inter-switch traffic. For AT&T, traffic is distributed from switch I to each switch J in proportion to the size of the trunk group from I to J. If the Late case is being calculated, then the calculation also reduces access traffic by each destination switch's proportion of unconnected EOs. For the Early case, traffic is allocated as if a normal connection existed to each destination.

3.2 Diagonal elements of SS represent intra-switch traffic proportions (e.g., the proportion of switch I's traffic that comes right back down to switch I). These quantities are calculated by multiplying the access traffic at a switch by PropIntra. As in 3.1, the Late case also involves reducing traffic in proportion to unreachable destinations.

3.3 At this point in the calculation, SS elements represent actual traffic values, not just proportions. The sum of each row of the SS matrix therefore represents the new access traffic for each switch, which would only be reduced from the original ACSlec if the Late case is being calculated. Each row of the SS matrix is set to this new access traffic value, ACSlec_dam.

3.4 To this point, the reduction in a switch's access traffic from ACSlec to ACSlec_dam has not been rolled all the way down to EO access traffic. In this step, the value F0dmg is set equal to AC for every EO, adjusted by the reduction from ACSlec to ACSlec_dam.

Stage 4: Calculate Effects of Regional Focused Overload

This stage calculates the network values that must change to reflect a regional focused overload, only in the event that `rtog` is true (indicating use of the `-r` option with a valid regional overload input file). There are three steps to this process.

4.1 The first step required is a call to `MakeACSRegion()` to calculate the `ACSRegion` array. This structure divides access traffic at each switch, `ACSlec_dam`, by region of origin. For example, if a switch serves EOs that are in two different regions, `ACSRegion` will hold the switch's access traffic from each region. This is required because the regional overload multiplier is in part a function of originating region.

4.2 The second step is a call to `MakeRegionalSS()` to calculate `SSReg`, the regional backbone traffic distribution. This step applies an overload from `RMultMatrix` to each normal `SS` element based on both origination and destination regions. ACSRegion is used from the previous step to perform traffic weighting. Note that this step does more than merely reproportion traffic—it increases it based on focused overload affects among regions.

4.3 The last step is a call to `MakeRegionalF0dmg()` to ensure that the overloads in the `SSReg` matrix get reflected back down to the EOs that originate the traffic. This will inflate access traffic at each EO based on average focused overload affects from the EO to every other destination in the network. The result is placed in `RegF0dmg`. The end-to-end traffic for any pair of end offices can now be calculated using a combination of `RegF0dmg` and `SSReg`.

Stage 5: Write output file

This stage involves calling OutWrite to write each of the 25 network description data structures to the multi-section output file. This file will be processed by `merge` before being used by `lecam`.

Inputs

character	keyfile[ ]	string containing the name of the key file	

Outputs

global	character[80]	eodmg_file	the name of the EO damage file
		atdmg_file	the name of the AT damage file
		eoatdmg_file	the name of the EO-AT span damage file
		regovl_file	the name of the regional overload file
		in_file	the name of the input network description created by attlive
		swdmg_file	the name of the IEC switch damage file
		popdmg_file	the name of the IEC POP damage file
		eopopdmg_file	the name of the EO-POP span damage file
		atpopdmg_file	the name of the AT-POP span damage file
		popswdmg_file	the name of the POP-switch span damage file
		qlink_file	the name of the qlink file

returns no formal values are returned

Purpose To read in the names of all necessary input files used by the main module

Called By main()

Calls To none

Local Variables

character	line[80]	temporarily holds a line of input from the keyfile
integer	att, mci spr	toggles used to indicate the presence of each IEC's data files in the keyfile

Global Variables see Outputs above for others

file	* fptr	general file pointer variable used to open keyfile

Algorithmic Description

This function opens and processes the TAMI keyfile, which contains the list of all input files required to perform a TAMI run. Refer to Appendix B (TAMI Keyfile description) for more detail on the keyfile.

LoadKeyfile processes the TAMI keyfile using the following four steps:

1. Skip past comment lines (lines starting with '#')
2. Process IEC string ('ATT/MCI/SPR') to make sure AT&T is included
3. Skip to LEC file section and read LEC damage file names
4. Skip to ATT section and read ATT file names

Inputs none

Outputs none

returns no formal values are returned

Purpose To initialize all of the network and traffic variables before they are read in from the input files

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	loop variables

Global Variables All global network and traffic variables. See main module description.

Algorithmic Description This function initializes all network and traffic variables. Traffic and trunk size quantities are initialized to 0. Damage vectors are initialized to 1 (survival). `Popswitch[ ]` is initialized to -1 to indicate no switch homed to each POP.

Inputs

file	`*inptr`	pointer to input IEC network file

Outputs Global network and traffic variables—described below under Algorithmic Description

returns no formal values are returned

Purpose To read in the undamaged network and traffic variables from the attlive output file

Called By `main()`

Calls To none

Local Variables

integer	`i,j,k,m,n,L,P,S`	loop variables
	`offset, count`	record position and counting variables
character	`line[ 100] , temp[ 15]`	buffer memory

Global Variables Global network and traffic variables—described below under Algorithmic Description.

character `template[.NUM_TEMP][ 15]`

constant `NUM_TEMP`

Algorithmic Description This function opens the undamaged network description created by attlive. It reads in each section of the file, as indicated by the section labels in the `template[ ]` array. The following global variables are loaded from the input file:

NUMSW	AC
NUMPOP	ACSlec / ACSlec_live
NUMAT	TAP
NUMEO	SAP
ATPOPSIZE	D3 / D3live
AT	ACPiec
HUPOP	popswitch
ATPOP	LATA
HAP	PhysDiv
SEP	IMT2
TEP	PropIntra
TEA	
F0	

Note that each of the 25 sections in the file may be stored in a different format and may have a different number of records depending on the variable type being read. For this reason, each section must be read in using the same formats used in attlive when the file was written out. The dimensions, `NUMSW, NUMPOP, NUMAT, NUMEO`, and `ATPOPSIZE`, are also used to determine the sizes of the sections that follow.

Inputs

character	`*infile`	input file

Outputs

returns no formal returns

Globals:

float	`Rmult`	regional traffic multiplier
	`Lmult`	LATA traffic multiplier
integer	`LReg`	LATA to region mapping
	`EOReg`	EO to region mapping

Purpose This routine loads regional information.

Called By `main()`

Calls To library functions
`print_info(void)`

Local Variables

double	`multiplier`	
character	`buffer1[ 80], buffer2[ 80]`	working memory
integer	`i,lata, region`	loop variable
FILE	`*inputfile`	input file

Global Variables

integer
`NUMREGIONS`
`NUMEO`
`EOlata`
`EO_SIZE`

Algorithmic Description `Loadoverloads()` first opens the indicated file. For each region, it then retrieves the region designation and multiplier. It then reads one line for each LATA in the region. The overload and region designation for the lata is then set. `Loadoverloads()` then closes the file, and sets the region for each EO.

Inputs none

Outputs

returns no formal returns

Globals:
float `RmultMatrix`

Purpose This routine generates a matrix of region to region traffic multipliers.

Called By `main()`

Calls To library functions

Local Variables

integer `i, j` loop variable

Global Variables

float `Rmult`

Algorithmic Description This routine builds a symmetric region to region traffic multiplier matrix. The region to region traffic multiplier is taken as the max of the two regional multipliers.

Inputs none

Outputs

global	float	`ACSRegion[ NUM_SWITCH][ NUM_REGION]`

returns no formal values are returned

Purpose To categorize switch access traffic by region of origin for a regional focused overload run

Called By `main()`

Calls To none

Local Variables

integer	`k`	loop variable

Global Variables

integer	`EOReg`	for every EO, indicates the region it is in
	`SW`	for every EO, indicates its unique homed IEC switch
float	`F0dmg`	post-damage engineered access traffic at an EO

Algorithmic Description

This function sums up F0dmg for each EO to its home switch based on the region, EOReg, in which the EO is located. Its purpose is to aggregate and track, at the IEC switch level, the originating region of all access traffic. This result is important for calculating the regional SS matrix in function `MakeRegionalSS()`.

Inputs none

Outputs

global	float	AvgSwRmult[NUM_SWITCH] [NUM_REGION]
	float	SSreg[NUM_SWITCH] [NUM_SWITCH]

returns no formal values are returned

Purpose To compute the affect of regional focused overload on the backbone traffic distribution, and to measure the traffic-weighted overload created by regional affects for each originating switch/region combination.

Called By main()

Calls To none

Local Variables

integer	i,j	loop variables on dimension NUMSW
	Ri, Rj	loop variables on dimension NUMREGION

Global Variables

integer	EOReg	for every EO, indicates the region it is in
	SW	for every EO, indicates its unique homed IEC switch
float	ACSlec_dam	for every switch, indicates post-damage access traffic before regional overload is assessed
	RmultMatrix	regional overload affect for every pair of originating and terminating regions
	ACSRegion	access traffic offered from each region through each switch
	SS	non-regional backbone traffic distribution

Algorithmic Description This function calculates the regional focused overload affect for each switch pair and stores the result in SSreg. For every switch pair, traffic is examined for every possible pair of regions between the switches. Each element of this traffic is subjected to overload defined by RmultMatrix. For example, from region Ri at switch i, to region Rj at switch j, traffic is overloaded by RmultMatrix[Ri][Rj] . These elements are weighted base on traffic at both ends, summed for all region combinations, and normalized to determine the average overload between the switch pair. This quantity, SStemp/ACSlec_dam, is multiplied by the original SS value to yield a regional overloaded SSreg value.

Note that an important intermediate result, AvgSwRmult, is calculated along the way. This value is the average focused-overload-induced multiplier applied to traffic leaving region Ri through switch i. It is computed from the overload experienced from region Ri at switch i to every other region/switch combination, Rj and j. This result has intentionally not been weighted by access traffic. This allows it to be used on a case-by-case basis for various access traffic values, as seen in function MakeRegionalF0dmg() .

Inputs none

Outputs

global	float	RegF0dmg[NUM_EO]
returns	no formal values are returned	

Purpose To compute the affect of regional focused overload on the backbone traffic distribution, and to measure the traffic-weighted overload created by regional affects for each originating switch/region combination.

Called By main()

Calls To none

Local Variables

integer	k	loop variable on dimension NUMEO

Global Variables

integer	EOReg	for every EO, indicates the region it is in
	SW	for every EO, indicates its unique homed IEC switch
float	F0dmg	for every EO, indicates the engineered access traffic prior to regional focused overload affects
	AvgSwRmult	indicates the traffic-weighted average regional focused overload experienced by access traffic from each region through each switch

Algorithmic Description This function calculates RegF0dmg for every EO. This quantity is a function of the post-damage engineered access traffic at the EO, F0dmg, and the average overload caused by regional overload effects, AvgSwRmult. AvgSwRmult, in turn, is a function of what region the EO resides in, and what switch it is homed to. For a regional focused overload, RegF0dmg represents the load offered at the EOs to the IEC.

Inputs

file	`*outptr`	pointer to output damaged IEC network file

Outputs Global network and traffic variables are written to file—described below under Algorithmic Description

returns no formal values are returned

Purpose To write out all the damaged network and traffic variables to the output file

Called By `main()`

Calls To none

Local Variables

integer	`k,l,m,n`	loop variables

Global Variables Global network and traffic variables—described below under Algorithmic Description.

Algorithmic Description This function writes a carrier-specific header to the output file, followed by the global damaged network and traffic variables. Each variable is written to a labeled section, with the following description:

`IECid`	identifies the carrier (att, mci, or spr)
`SWITCH_SIZE`	number of IEC switches in the network
`POP_SIZE`	number of POPs in the network
`AT_SIZE`	number of ATs in the network
`EO_SIZE`	number of EOs in the network
`ATPOPSIZE`	maximum number of POPs an AT homes to for this network
`EOAT`	for every EO, identifies the homed AT (-1 if N/A)
`HUPOP`	for every EO, identifies the homed HU POP
`ATPOP`	for every EO, identifies the homed POP through the AT
`HAP`	for every AT, identifies all homed POPs
`SD`	for every EO, identifies the size of the EO-POP TG
`TEA`	for every EO, identifies the total (both ways) traffic engineered for the EO-AT TG
`F0`	for every EO, identifies the components of access traffic offered to the HU and final homings respectively
`F0dmg`	for every EO, identifies the amount of engineered access traffic offered by the EO after damage affects are considered
`ELAP`	for every AT, identifies the total AT-POP traffic for each AT-POP homing defined in HAP
`SAP`	for every AT, identifies the size of each AT-POP TG
`ACS`	for every IEC switch, identifies the total access traffic offered for inter-switch transport
`POPSWITCH`	for every POP, identifies the homed IEC switch
`AC1`	for every EO, identifies the actual traffic offered, including regional focused overload, if applicable

`EOalive`	for every EO, indicates 0/1 for failure/survival
`ATalive`	for every AT, indicates 0/1 for failure/survival
`POPalive`	for every POP, indicates 0/1 for failure/survival
`SWalive`	for every switch, indicates 0/1 for failure/survival
`EOlata`	for every EO, identifies its LATA
`PhysDiv`	for every EO, identifies if the HU homing (if any) is physically diverse from the final homing (if any)
`SS`	for the matrix of IEC switch pairs, identifies traffic distribution from one switch to the other
`SSreg`	same as SS, but for regional focused overload
`PropIntra`	for every switch, indicates the proportion of access traffic that is intra-switch traffic

Note that each of the sections in the file may be stored in a different format and may have a different number of records depending on the variable type being written.

3.8 mciwdmg: Pre-process Damaged MCI Network and Traffic

Purpose This module performs the data preparation calculations required to assess the affects of damage to the LEC homing chains and end-to-end traffic matrix for the MCI network. Many of the engineered traffic variables computed by `mcilive` are adjusted to reflect post-damage engineered traffic under Early and Late scenarios. For a regional analysis, this module computes the focused overload offered load matrix through MCI. The output file from this module is merged with the other IEC files in the `merge` module before being input into the `lecam` module.

Call Syntax `mciwdmg -k <key file> [ options]`

mandatory:

`-d`	use damage vector `<number>` for this run
`-k`	reads in input file `<key file>`

options:

`-b`	blast-type damage: use physical diversity model for HU damage
`-e`	specify early scenario <1> or <2>
`-i`	no longer used
`-r`	reads in regional overload input file `<reg file>`
`-s`	specify non-default random number generator stream using `<stream>` from 1 to 15
`-?`	user help—prints call syntax and exits without running

example `mciwdmg -k tamirun.key -d5`

Input Files The list of input files required by mciwdmg is supplied in the keyfile. See Appendix B for these files.

reg file In addition to the files in the keyfile, for a regional overload run, the -r option will specify a regional overload file, <reg file>. The first line of this file specifies the default (Region 0) overload. Unless specified in later regions, all LATAs will be overloaded based on this default. The second line defines Region 1 and its overload value. The next line contains a list of space-separated LATA numbers that are members of Region 1. Subsequent regions are defined similarly, up to a maximum of five regions.

format	R0, <region overload>	default region line
	<Region number>, <region overload> (c2, 1x, f)	subsequent region line
	<LATA number>, . . . , <LATA number> (i3, 1x, . . ., i3, 1x, i3)	with associated LATAs
example	`R0 1.25`	defines 1.25 overload for default LATAs
	`R1 5.0`	defines 5.0 overload for region 1 LATAs
	`132 133 134`	region 1 LATAs defined here as a list
	`R2 10.0`	defines 10.0 overload for region 2 LATAs
	`251 291 237 252`	region 2 LATAs defined here as a list

ppmeanfile Additionally, mciwdmg reads in a file with the hardcoded name "ppmeanfile.mci." This file contains a matrix of switch to switch blockages for MCI which indicate whether a connection exists between two switches given the routing rules and damage scenario. This file is created

by qmod_mci. Each line contains 12 row elements of the switch-to-switch blockage matrix, and the row will wrap as many lines as necessary. If a row ends midway through a line, the rest of that line is blank. The next row of the matrix will begin on a new line until the entire matrix has been printed. Elements are separated by spaces within a line.

Output Files

mci2lecam.out--multisection output file described in function Outwrite

Includes

`"fileio.c"`	includes input/output utility functions—see Appendix A
`"lecam.h"`	defines lecam constants and variables—see Section 3.11
`"waglib.h"`	defines random number generator functions, constants, and data structures—not used

Constants

`MAX_BIN 3`	Not used
`NUM_TEMP 25`	Number of sections in output file

constants defined in lecam.h (network sizing variables) and used in mciwdmg:

`TRUE`	`1`	
`FALSE`	`0`	
`MAX_AT`	`1000`	Maximum number of ATs in network
`MAX_EO`	`19250`	Maximum number of EOs in network
`ATPOPMAX`	`9`	Maximum number of POPs a single AT may home to
`MAX_POP`	`575`	Maximum number of POPs in network
`MAX_SWITCH`	`130`	Maximum number of IEC switches in network
`MAX_LATA`	`1000`	Maximum number of LATAs
`MAX_REGION`	`5`	Maximum number of regions defined for regional overload

Global Variables

FILE	`*inptr`	File pointer for an input file
	`*outptr`	File pointer for the output file
	`*fptr`	File pointer for any file
int	`KEYTOG`	toggle for -k option
	`rtog`	toggle for -r option
	`NUMEO`	number of EOs in network
	`NUMAT`	number of ATs in network
	`NUMPOP`	number of POPs in network
	`NUMSW`	number of backbone switches in network
	`NUMREGIONS`	number of regions defined for regional overload
	`ATPOPSIZE`	maximum number of AT-POP homing segments
	`HAP[ MAX_AT][ ATPOPMAX]`	AT-POP homings
	`AT[ MAX_EO]`	homed AT for an EO
	`ATPOP[ MAX_EO]`	homed POP through the AT for an EO
	`HUPOP[ MAX_EO]`	homed high-usage POP for an EO
	`SW[ MAX_EO]`	homed IEC switch for an EO
	`TYPE[ MAX_EO]`	homing type for an EO
	`LATA[ MAX_EO]`	LATA designation for an EO
	`popswitch[ MAX_POP]`	homed IEC switch for POP
	`IMT[ MAX_SWITCH][ MAX_SWITCH]`	matrix of IMT TGs between IEC switches
	`IMT2[ MAX_SWITCH][ MAX_SWITCH]`	copy of IMT matrix used for data checking purposes

	IMTdam[MAX_SWITCH] [MAX_SWITCH]	damaged IMT matrix
	SUM_IMT[MAX_SWITCH]	total IMT TGs originating or terminating at each IEC switch
	SAP[MAX_AT] [ATPOPMAX]	size of AT-POP TG for each POP homed to AT
	SEP[MAX_EO]	size of the direct EO-POP TG, if exists
	PhysDiv[MAX_EO]	0/1 indication of whether the HU TG, if existing, is physically diverse from the AT
	EOalive[MAX_EO]	0/1 indication of survival of EO
	ATalive[MAX_AT]	0/1 indication of survival of AT
	POPalive[MAX_POP]	0/1 indication of survival of POP
	SWalive[MAX_SW]	0/1 indication of survival of IEC switch
	EOATalive[MAX_EO]	0/1 indication of survival of EO-AT span
	ATPOPalive[MAX_AT] [ATPOPMAX]	0/1 indication of survival of AT-POP span
	EOPOPalive[MAX_EO]	0/1 indication of survival of EO-POP span
	POPSWalive[MAX_POP]	0/1 indication of survival of POP-SW span
	LReg[MAX_LATA]	Indicates which Region each LATA is in
	EOReg[MAX_EO]	Indicates which Region each EO is in
float	TAP[MAX_AT] [ATPOPMAX]	traffic engineered for SAP TGs
	TEP[MAX_EO]	traffic engineered for SEP TGs
	TEA[MAX_EO]	traffic engineered between EO and AT
	ACSlec[MAX_SWITCH]	traffic offered from LEC to each IEC switch
	ACSlec_live[MAX_SWITCH]	traffic offered from LEC to each IEC switch without considering network damage
	ACSlec_dam[MAX_SWITCH]	traffic offered from LEC to each IEC switch including effects of damage
	ACSiec[MAX_SWITCH]	traffic offered to IEC switch for inter-switch transport
	ACStest[MAX_SWITCH]	temporary testing of ACS
	PropIntra[MAX_SWITCH]	proportion of inter-switch traffic at each IEC switch
	D3[MAX_POP]	traffic offered from LEC to each POP
	D3live[MAX_POP]	traffic offered from LEC to each POP in the ase of no network damage
	D3test[MAX_POP]	temporary testing of D3
	ACPiec[MAX_POP]	traffic through each POP engineered for inter-switch transport
	AC[MAX_EO]	traffic offered from each EO
	Morph[MAX_SWITCH] [MAX_SWITCH] , MorphRow[MAX_SWITCH]	holds the proportions matrix elements, and a row sum respectively, read in from the traffic distribution input file
	SWBLOCK[MAX_SWITCH] [MAX_SWITCH]	holds the switch-to-switch connectivity results read in from ppmeanfile.mci
	Lmult[MAX_LATA]	indicates regional overload multiplier for each LATA
	Rmult[MAX_REGION]	indicates regional overload multiplier assigned to each region
	ACSRegion[MAX_SWITCH] [MAX_REGION]	indicates traffic offered to each switch from each region
	AvgSwRmult[MAX_SWITCH] [MAX_REGION]	indicates the traffic-weighted average multiplier applied to each element of ACSRegion

	`RmultMatrix[ MAX_REGION] [ MAX_REGION]`	indicates the traffic multiplier applied to traffic from one region to another
double	`SS[ MAX_SWITCH] [ MAX_SWITCH]`	the main switch-to-switch proportions matrix that describes how traffic is distributed across the IEC backbone
	`SSreg[ MAX_SWITCH] [ MAX_SWITCH]`	the SS matrix after being modified to reflect regional focused overload
	`SSrow[ MAX_SWITCH]`	sum of an SS row, used for normalizing the matrix
	`F0[ MAX_EO] [ 2]`	Engineered access traffic at an EO divided by HU and final engineered proportions
	`F0dmg[ MAX_EO]`	Engineered access traffic at an EO with damage considered
	`RegF0dmg[ MAX_EO]`	Engineered access traffic at an EO with damage and regional overload considered
character	`template[ NUM_TEMP] [ 15]`	holds the labels for each section of the output file
	`in_file[ 80]`	name of network file output by `mcilive`
	`qlink_file[ 80]`	name of qlink file
	`morph_file[ 80]`	name of traffic distribution file
	`eodmg_file[ 80]`	name of EO damage file
	`atdmg_file[ 80]`	name of AT damage file
	`popdmg_file[ 80]`	name of POP damage file
	`swdmg_file[ 80]`	name of IEC switch damage file
	`eoatdmg_file[ 80]`	name of EO-AT span damage file
	`eopopdmg_file[ 80]`	name of EO-POP high usage span damage file
	`atpopdmg_file[ 80]`	name of AT-POP span damage file
	`popswdmg_file[ 80]`	name of POP-switch span damage file
	`regovl_file[ 80]`	name of regional overload file, if used

Local Variables

Variables local to `main()`:

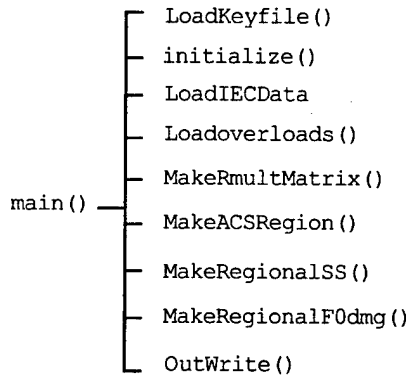
extern	integer `optind, argc`	command line options counters
	character `*optarg,**argv[ ]`	reference to command line option string
char	`c`	general single character variable
	`out_file[ 80]`	name of output file
	`keyfile[ 80]`	name of keyfile
	`line[ 200]`	buffer for line of input from file
	`temp[ 5]`	temporary text buffer
int	`err`	toggle to indicate error in command line arguments
	`etog`	toggle to indicate -e option
	`stog`	toggle to indicate -s option
	`ttog`	toggle to indicate -t option
	`itog`	toggle to indicate -i option
	`otog`	toggle to indicate -o option
	`qtog`	toggle to indicate -q option
	`dtog`	toggle to indicate -d option
	`btog`	toggle to indicate -b option
	`stream`	random number stream (default = 10)

	dvector	damage vector number to use (default = 1)
	early	indicates Early 1 or 2 cases (default = 0)
	done	used as a boolean in loops
	num_dead, num_tot	counts node and span damage summary statistics
	cap_live, cap_tot, cap	counts IEC TG damage summary statistics
	i,j,k,l,pm,p,m,n	general loop count variables
	test,surv,dummy	general temporary variables
float	sum_traf, sum_reg_traf	variables to count total traffic in the network

Component Functions

LoadKeyfile()	Loads AT&T input file list from Keyfile
initialize()	Initializes all network and traffic variables
LoadIECData()	Loads input network file (output from mcilive)
Loadoverloads()	Loads regional overload input file
MakeRmultMatrix()	Calculates the RmultMatrix structure
MakeACSRegion()	Calculates the ACSRegion value
MakeRegionalSS()	Calculates the backbone distribution matrix for regional overload
MakeRegionalF0dmg()	Calculates the EO access traffic for regional overload
OutWrite()	Writes network and traffic variables to output file

Function Tree



Algorithmic Description

Module mciwdmg is used to apply a given damage scenario to the undamaged MCI network description and traffic matrix. In almost every aspect, it is the same as the attwdmg module. The differences from attwdmg are described here, referencing the attwdmg module documentation where appropriate.

For attwdmg, it was assumed that backbone traffic was distributed in proportion to the trunks. In MCI, this assumptions does not hold. Therefore, mciwdmg requires an additional input file, which the mciwdmg code refers to as the morph file, to describe the undamaged traffic distribution. Refer to mcilive module documentation for more information on the calculation of this traffic distribution file.

In addition, mciwdmg requires the determination of connectivity between an pair of backbone switches. For AT&T and Sprint, which are assumed to have direct trunking between all switch pairs, the IMTdam parameter was sufficient. For MCI, which uses hierarchical routing, this parameter cannot be used. Instead, a "trickle current" qtcm module run is performed. This run uses the minimum allowed offered traffic matrix between switches so as not to cause blocking due to congestion. If there is no route for traffic between two switches, the blocking will still

show up as 1.0, indicating no connectivity. This information is read in to the SWBLOCK[i][j] variable from the ppmeanfile.mci file output by qtcm.

Both the morph file and ppmeanfile.mci are applied in the calculation of SS. Where attwdmg uses IMT[i][j] /SUM_IMT[i] to distribute switch i's traffic to all other switches j, mciwdmg replaces this term with Morph[i][j] /MorphRow[i] . In addition, code supporting the SS calculation uses the term MorphRow[i] instead of the term SUM_IMT[i] . The ppmeanfile information is used for the Late case calculation of SS, where it must be determined whether a connection exists between a pair of switches. The calculation for mciwdmg replaces the term IMTdam[i] ==0 with the term SWBLOCK[i] ==1.0 in the "if" clause that determines connectivity between switches.

In all other respects, mciwdmg operates identically to attwdmg.

Inputs

character	keyfile[ ]		string containing the name of the key file

Outputs

global	character[80]	eodmg_file	the name of the EO damage file
		atdmg_file	the name of the AT damage file
		eoatdmg_file	the name of the EO-AT span damage file
		regovl_file	the name of the regional overload file
		in_file	the name of the input network description created by attlive
		swdmg_file	the name of the IEC switch damage file
		popdmg_file	the name of the IEC POP damage file
		eopopdmg_file	the name of the EO-POP span damage file
		atpopdmg_file	the name of the AT-POP span damage file
		popswdmg_file	the name of the POP-switch span damage file
		qlink_file	the name of the qlink file
		morph_file	the name of the IEC traffic distribution file

returns no formal values are returned

Purpose To read in the names of all necessary input files used by the main module

Called By main()

Calls To none

Local Variables

character	line[80]	temporarily holds a line of input from the keyfile
integer	att, mci	toggles used to indicate the presence of each IEC's data files in
	spr	the keyfile

Global Variables see Outputs above for others

file	*fptr	general file pointer variable used to open keyfile

Algorithmic Description

This function opens and processes the TAMI keyfile, which contains the list of all input files required to perform a TAMI run. Refer to Appendix B (TAMI Keyfile description) for more detail on the keyfile.

LoadKeyfile processes the TAMI keyfile using the following four steps:

1. Skip past comment lines (lines starting with '#')
2. Process IEC string ('ATT/MCI/SPR') to make sure MCI is included
3. Skip to LEC file section and read LEC damage file names
4. Skip to MCI section and read MCI input file names

Inputs none

Outputs none

returns no formal values are returned

Purpose To initialize all of the network and traffic variables before they are read in from the input files

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	loop variables

Global Variables All global network and traffic variables. See main module description.

Algorithmic Description This function initializes all network and traffic variables. Traffic and trunk size quantities are initialized to 0. Damage vectors are initialized to 1 (survival). `Popswitch[ ]` is initialized to -1 to indicate no switch homed to each POP.

Inputs

file	`*inptr`	pointer to input IEC network file

Outputs Global network and traffic variables—described below under Algorithmic Description

returns no formal values are returned

Purpose To read in the undamaged network and traffic variables from the mcilive output file

Called By `main()`

Calls To none

Local Variables

integer	`i,j,k,m,n,L,P,S`	loop variables
	`offset, count`	record position and counting variables
character	`line[ 100], temp[ 15]`	buffer memory

Global Variables Global network and traffic variables—described below under Algorithmic Description.

char	`template[ NUM_TEMP][ 15]`
constant	`NUM_TEMP`

Algorithmic Description This function opens the undamaged network description created by mcilive. It reads in each section of the file, as indicated by the section labels in the `template[ ]` array. The following global variables are loaded from the input file:

NUMSW	AC
NUMPOP	ACSlec / ACSlec_live
NUMAT	TAP
NUMEO	SAP
ATPOPSIZE	D3 / D3live
AT	ACPiec
HUPOP	popswitch
ATPOP	LATA
HAP	PhysDiv
SEP	IMT2
TEP	PropIntra
TEA	
F0	

Note that each of the 25 sections in the file may be stored in a different format and may have a different number of records depending on the variable type being read. For this reason, each section must be read in using the same formats used in mcilive when the file was written out. The dimensions, `NUMSW, NUMPOP, NUMAT, NUMEO`, and `ATPOPSIZE`, are also used to determine the sizes of the sections that follow.

Inputs

character	`*infile`	input file

Outputs

returns: no formal returns

Globals:

real	`Rmult`	regional traffic multiplier
	`Lmult`	LATA traffic multiplier
integer	`LReg`	LATA to region mapping
	`EOReg`	EO to region mapping

Purpose This routine loads regional information.

Called By `main()`

Calls To library functions
`print_info(void)`

Local Variables

double	`multiplier`	
character	`buffer1[80], buffer2[80]`	working memory
integer	`i,lata, region`	loop variable
FILE	`*inputfile`	input file

Global Variables

integer
`NUMREGIONS`
`NUMEO`
`EOlata`
`EO_SIZE`

Algorithmic Description `Loadoverloads()` first opens the indicated file. For each region, it then retrieves the region designation and multiplier. It then reads one line for each LATA in the region. The overload and region designation for the lata is then set. `Loadoverloads()` then closes the file, and sets the region for each EO.

Inputs none

Outputs

returns no formal returns

Globals:
real `RmultMatrix`

Purpose This routine generates a matrix of region to region traffic multipliers..

Called By `main()`

Calls To library functions

Local Variables

integer `i, j` loop variable

Global Variables
real `Rmult`

Algorithmic Description This routine builds a symmetric region to region traffic multiplier matrix. The region to region traffic multiplier is taken as the max of the two regional multipliers.

Inputs none

Outputs

global	float	`ACSRegion[ NUM_SWITCH][ NUM_REGION]`
returns	no formal values are returned	

Purpose To categorize switch access traffic by region of origin for a regional focused overload run

Called By `main()`

Calls To none

Local Variables

integer	`k`	loop variable

Global Variables

integer	`EOReg`	for every EO, indicates the region it is in
	`SW`	for every EO, indicates its unique homed IEC switch
float	`F0dmg`	post-damage engineered access traffic at an EO

Algorithmic Description This function sums up F0dmg for each EO to its home switch based on the region, EOReg, in which the EO is located. Its purpose is to aggregate and track, at the IEC switch level, the originating region of all access traffic. This result is important for calculating the regional SS matrix in function `MakeRegionalSS()`.

Inputs none

Outputs

global	float	AvgSwRmult[NUM_SWITCH][NUM_REGION]
	float	SSreg[NUM_SWITCH][NUM_SWITCH]

returns no formal values are returned

Purpose To compute the affect of regional focused overload on the backbone traffic distribution, and to measure the traffic-weighted overload created by regional affects for each originating switch/region combination.

Called By main()

Calls To none

Local Variables

integer	i,j	loop variables on dimension NUMSW
	Ri, Rj	loop variables on dimension NUMREGION

Global Variables

int	EOReg	for every EO, indicates the region it is in
	SW	for every EO, indicates its unique homed IEC switch
float	ACSlec_dam	for every switch, indicates post-damage access traffic before regional overload is assessed
	RmultMatrix	regional overload affect for every pair of originating and terminating regions
	ACSRegion	access traffic offered from each region through each switch
	SS	non-regional backbone traffic distribution

Algorithmic Description This function calculates the regional focused overload affect for each switch pair and stores the result in SSreg. For every switch pair, traffic is examined for every possible pair of regions between the switches. Each element of this traffic is subjected to overload defined by RmultMatrix. For example, from region Ri at switch i, to region Rj at switch j, traffic is overloaded by RmultMatrix[Ri][Rj] . These elements are weighted base on traffic at both ends, summed for all region combinations, and normalized to determine the average overload between the switch pair. This quantity, SStemp/ACSlec_dam, is multiplied by the original SS value to yield a regional overloaded SSreg value.

Note that an important intermediate result, AvgSwRmult, is calculated along the way. This value is the average focused-overload-induced multiplier applied to traffic leaving region Ri through switch i. It is computed from the overload experienced from region Ri at switch i to every other region/switch combination, Rj and j. This result has intentionally not been weighted by access traffic. This allows it to be used on a case-by-case basis for various access traffic values, as seen in function MakeRegionalF0dmg() .

Inputs none

Outputs

global	float	RegF0dmg[NUM_EO]
returns	no formal values are returned	

Purpose To compute the affect of regional focused overload on the backbone traffic distribution, and to measure the traffic-weighted overload created by regional affects for each originating switch/region combination.

Called By main()

Calls To none

Local Variables

integer	k	loop variable on dimension NUMEO

Global Variables

int	EOReg	for every EO, indicates the region it is in
	SW	for every EO, indicates its unique homed IEC switch
float	F0dmg	for every EO, indicates the engineered access traffic prior to regional focused overload affects
	AvgSwRmult	indicates the traffic-weighted average regional focused overload experienced by access traffic from each region through each switch

Algorithmic Description This function calculates RegF0dmg for every EO. This quantity is a function of the post-damage engineered access traffic at the EO, F0dmg, and the average overload caused by regional overload effects, AvgSwRmult. AvgSwRmult, in turn, is a function of what region the EO resides in, and what switch it is homed to. For a regional focused overload, RegF0dmg represents the load offered at the EOs to the IEC.

Inputs

file	`*outptr`	pointer to output damaged IEC network file

Outputs Global network and traffic variables are written to file—described below under Algorithmic Description

returns no formal values are returned

Purpose To write out all the damaged network and traffic variables to the output file

Called By `main()`

Calls To none

Local Variables

integer	`k,l,m,n`	loop variables

Global Variables Global network and traffic variables—described below under Algorithmic Description.

Algorithmic Description This function writes a carrier-specific header to the output file, followed by the global damaged network and traffic variables. Each variable is written to a labeled section, with the following description:

`IECid`	identifies the carrier (att, mci, or spr)
`SWITCH_SIZE`	number of IEC switches in the network
`POP_SIZE`	number of POPs in the network
`AT_SIZE`	number of ATs in the network
`EO_SIZE`	number of EOs in the network
`ATPOPSIZE`	maximum number of POPs an AT homes to for this network
`EOAT`	for every EO, identifies the homed AT (-1 if N/A)
`HUPOP`	for every EO, identifies the homed HU POP
`ATPOP`	for every EO, identifies the homed POP through the AT
`HAP`	for every AT, identifies all homed POPs
`SD`	for every EO, identifies the size of the EO-POP TG
`TEA`	for every EO, identifies the total (both ways) traffic engineered for the EO-AT TG
`F0`	for every EO, identifies the components of access traffic offered to the HU and final homings respectively
`F0dmg`	for every EO, identifies the amount of engineered access traffic offered by the EO after damage affects are considered
`ELAP`	for every AT, identifies the total AT-POP traffic for each AT-POP homing defined in HAP
`SAP`	for every AT, identifies the size of each AT-POP TG
`ACS`	for every IEC switch, identifies the total access traffic offered for inter-switch transport
`POPSWITCH`	for every POP, identifies the homed IEC switch
`AC1`	for every EO, identifies the actual traffic offered, including regional focused overload, if applicable

EOalive	for every EO, indicates 0/1 for failure/survival
ATalive	for every AT, indicates 0/1 for failure/survival
POPalive	for every POP, indicates 0/1 for failure/survival
SWalive	for every switch, indicates 0/1 for failure/survival
EOlata	for every EO, identifies its LATA
PhysDiv	for every EO, identifies if the HU homing (if any) is physically diverse from the final homing (if any)
SS	for the matrix of IEC switch pairs, identifies traffic distribution from one switch to the other
SSreg	same as SS, but for regional focused overload
PropIntra	for every switch, indicates the proportion of access traffic that is intra-switch traffic

Note that each of the sections in the file may be stored in a different format and may have a different number of records depending on the variable type being written.

3.9 sprwdmg: Pre-process Damaged Sprint Network and Traffic

Purpose

This module performs the data preparation calculations required to assess the affects of damage to the LEC homing chains and end-to-end traffic matrix for the Sprint network. Many of the engineered traffic variables computed by `sprlive` are adjusted to reflect post-damage engineered traffic under Early and Late scenarios. For a regional analysis, this module computes the focused overload offered load matrix through Sprint. The output file from this module is merged with the other IEC files in the `merge` module before being input into the `lecam` module.

Call Syntax

`sprwdmg -k <key file> [ options]`

mandatory:

`-d`	use damage vector `<number>` for this run
`-k`	reads in input file `<key file>`

options:

`-b`	blast-type damage: use physical diversity model for HU damage
`-e`	specify early scenario <1> or <2>
`-i`	no longer used
`-r`	reads in regional overload input file `<reg file>`
`-s`	specify non-default random number generator stream using `<stream>` from 1 to 15
`-?`	user help—prints call syntax and exits without running

example

```
sprwdmg -k tamirun.key -d5
```

Input Files

The list of input files required by sprwdmg is supplied in the keyfile. See Appendix B (TAMI keyfile description) for these files.

reg file

In addition to the files in the keyfile, for a regional overload run, the -r option will specify a regional overload file, <reg file>. The first line of this file specifies the default (Region 0) overload. Unless specified in later regions, all LATAs will be overloaded based on this default. The second line defines Region 1 and its overload value. The next line contains a list of space-separated LATA numbers that are members of Region 1. Subsequent regions are defined similarly, up to a maximum of five regions.

format

R0, <region overload>	default region line
<Region number>, <region overload>	subsequent region line
(c2, 1x, f)	
<LATA number>, . . . , <LATA number>	with associated LATAs
(i3, 1x, . . ., i3, 1x, i3)	

example

```
R0 1.25                 defines 1.25 overload for default LATAs
R1 5.0                  defines 5.0 overload for region 1 LATAs
132 133 134             region 1 LATAs defined here as a list
R2 10.0                 defines 10.0 overload for region 2 LATAs
251 291 237 252         region 2 LATAs defined here as a list
```

Output Files

spr2lecam.out—multisection output file described in function Outwrite

Includes

"fileio.c"	includes input/output utility functions—see Appendix A
"lecam.h"	defines lecam constants and variables—see Section 3.11
"waglib.h"	defines random number generator functions, constants, and data structures—not used

Constants

MAX_BIN 3	Not used
NUM_TEMP 25	Number of sections in output file

constants defined in lecam.h (network sizing variables) and used in sprwdmg:

TRUE	1	
FALSE	0	
MAX_AT	1000	Maximum number of ATs in network
MAX_EO	19250	Maximum number of EOs in network
ATPOPMAX	9	Maximum number of POPs a single AT may home to
MAX_POP	575	Maximum number of POPs in network
MAX_SWITCH	130	Maximum number of IEC switches in network
MAX_LATA	1000	Maximum number of LATAs
MAX_REGION	5	Maximum number of regions defined for regional overload

Global Variables

FILE	*inptr	File pointer for an input file
	*outptr	File pointer for the output file
	*fptr	File pointer for any file
integer	KEYTOG	toggle for -k option
	rtog	toggle for -r option
	NUMEO	number of EOs in network
	NUMAT	number of ATs in network
	NUMPOP	number of POPs in network
	NUMSW	number of backbone switches in network
	NUMREGIONS	number of regions defined for regional overload
	ATPOPSIZE	maximum number of AT-POP homing segments
	HAP[MAX_AT] [ATPOPMAX]	AT-POP homings
	AT[MAX_EO]	homed AT for an EO
	ATPOP[MAX_EO]	homed POP through the AT for an EO
	HUPOP[MAX_EO]	homed high-usage POP for an EO
	SW[MAX_EO]	homed IEC switch for an EO
	TYPE[MAX_EO]	homing type for an EO
	LATA[MAX_EO]	LATA designation for an EO
	popswitch[MAX_POP]	homed IEC switch for POP
	IMT[MAX_SWITCH] [MAX_SWITCH]	matrix of IMT TGs between IEC switches
	IMT2[MAX_SWITCH] [MAX_SWITCH]	copy of IMT matrix used for data checking purposes
	IMTdam[MAX_SWITCH] [MAX_SWITCH]	damaged IMT matrix
	SUM_IMT[MAX_SWITCH]	total IMT TGs originating or terminating at each IEC switch
	SAP[MAX_AT] [ATPOPMAX]	size of AT-POP TG for each POP homed to AT

	SEP[MAX_EO]	size of the direct EO-POP TG, if exists
	PhysDiv[MAX_EO]	0/1 indication of whether the HU TG, if existing, is physically diverse from the AT
	EOalive[MAX_EO]	0/1 indication of survival of EO
	ATalive[MAX_AT]	0/1 indication of survival of AT
	POPalive[MAX_POP]	0/1 indication of survival of POP
	SWalive[MAX_SW]	0/1 indication of survival of IEC switch
	EOATalive[MAX_EO]	0/1 indication of survival of EO-AT span
	ATPOPalive[MAX_AT][ATPOPMAX]	0/1 indication of survival of AT-POP span
	EOPOPalive[MAX_EO]	0/1 indication of survival of EO-POP span
	POPSWalive[MAX_POP]	0/1 indication of survival of POP-SW span
	LReg[MAX_LATA]	Indicates which Region each LATA is in
	EOReg[MAX_EO]	Indicates which Region each EO is in
float	TAP[MAX_AT][ATPOPMAX]	traffic engineered for SAP TGs
	TEP[MAX_EO]	traffic engineered for SEP TGs
	TEA[MAX_EO]	traffic engineered between EO and AT
	ACSlec[MAX_SWITCH]	traffic offered from LEC to each IEC switch
	ACSlec_live[MAX_SWITCH]	traffic offered from LEC to each IEC switch without considering network damage
	ACSlec_dam[MAX_SWITCH]	traffic offered from LEC to each IEC switch including effects of damage
	ACSiec[MAX_SWITCH]	traffic offered to IEC switch for inter-switch transport
	ACStest[MAX_SWITCH]	temporary testing of ACS
	PropIntra[MAX_SWITCH]	proportion of inter-switch traffic at each IEC switch
	D3[MAX_POP]	traffic offered from LEC to each POP
	D3live[MAX_POP]	traffic offered from LEC to each POP in the case of no network damage
	D3test[MAX_POP]	temporary testing of D3
	ACPiec[MAX_POP]	traffic through each POP engineered for inter-switch transport
	AC[MAX_EO]	traffic offered from each EO
	Morph[MAX_SWITCH][MAX_SWITCH] , MorphRow[MAX_SWITCH]	holds the proportions matrix elements, and a row sum respectively, read in from the traffic distribution input file
	Lmult[MAX_LATA]	indicates regional overload multiplier for each LATA
	Rmult[MAX_REGION]	indicates regional overload multiplier assigned to each region
	ACSRegion[MAX_SWITCH][MAX_REGION]	indicates traffic offered to each switch from each region
	AvgSwRmult[MAX_SWITCH][MAX_REGION]	indicates the traffic-weighted average multiplier applied to each element of ACSRegion
	RmultMatrix[MAX_REGION][MAX_REGION]	indicates the traffic multiplier applied to traffic from one region to another
double	SS[MAX_SWITCH][MAX_SWITCH]	the main switch-to-switch proportions matrix that describes how traffic is distributed across the IEC backbone
	SSreg[MAX_SWITCH][MAX_SWITCH]	the SS matrix after being modified to reflect regional focused overload
	SSrow[MAX_SWITCH]	sum of an SS row, used for normalizing the matrix
	F0[MAX_EO][2]	Engineered access traffic at an EO divided by HU and final engineered proportions

	`FOdmg[ MAX_EO]`	Engineered access traffic at an EO with damage considered
	`RegFOdmg[ MAX_EO]`	Engineered access traffic at an EO with damage and regional overload considered
character	`template[ NUM_TEMP][ 15]`	holds the labels for each section of the output file
	`in_file[ 80]`	name of network file output by `sprlive`
	`qlink_file[ 80]`	name of qlink file
	`morph_file[ 80]`	name of traffic distribution file
	`eodmg_file[ 80]`	name of EO damage file
	`atdmg_file[ 80]`	name of AT damage file
	`popdmg_file[ 80]`	name of POP damage file
	`swdmg_file[ 80]`	name of IEC switch damage file
	`eoatdmg_file[ 80]`	name of EO-AT span damage file
	`eopopdmg_file[ 80]`	name of EO-POP high usage span damage file
	`atpopdmg_file[ 80]`	name of AT-POP span damage file
	`popswdmg_file[ 80]`	name of POP-switch span damage file
	`regovl_file[ 80]`	name of regional overload file, if used

Local Variables

Variables local to `main()`:

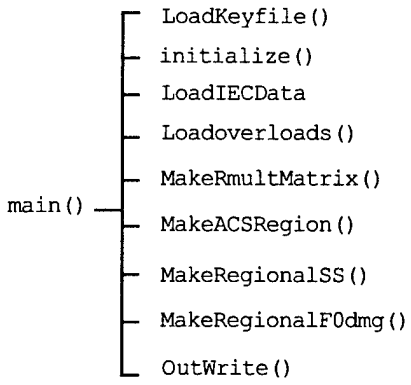
extern	integer `optind, argc`	command line options counters
	character `*optarg, ** argv[ ]`	reference to command line option string
character	`c`	general single character variable
	`out_file[ 80]`	name of output file
	`keyfile[ 80]`	name of keyfile
	`line[ 200]`	buffer for line of input from file
	`temp[ 5]`	temporary text buffer
integer	`err`	toggle to indicate error in command line arguments
	`etog`	toggle to indicate -e option
	`stog`	toggle to indicate -s option
	`ttog`	toggle to indicate -t option
	`itog`	toggle to indicate -i option
	`otog`	toggle to indicate -o option
	`qtog`	toggle to indicate -q option
	`dtog`	toggle to indicate -d option
	`btog`	toggle to indicate -b option
	`stream`	random number stream (default = 10)
	`dvector`	damage vector number to use (default = 1)
	`early`	indicates Early 1 or 2 cases (default = 0)
	`done`	used as a boolean in loops
	`num_dead, num_tot`	counts node and span damage summary statistics
	`cap_live, cap_tot, cap`	counts IEC TG damage summary statistics
	`i,j,k,l,pm,p,m,n`	general loop count variables
	`test,surv,dummy`	general temporary variables
float	`sum_traf, sum_reg_traf`	variables to count total traffic in the network

Component Functions

`LoadKeyfile()`	Loads AT&T input file list from Keyfile
`initialize()`	Initializes all network and traffic variables

`LoadIECData()`	Loads input network file (output from sprlive)
`Loadoverloads()`	Loads regional overload input file
`MakeRmultMatrix()`	Calculates the RmultMatrix structure
`MakeACSRegion()`	Calculates the ACSRegion value
`MakeRegionalSS()`	Calculates the backbone distribution matrix for regional overload
`MakeRegionalF0dmg()`	Calculates the EO access traffic for regional overload
`OutWrite()`	Writes network and traffic variables to output file

Function Tree



Algorithmic Description

Module `sprwdmg` is used to apply a given damage scenario to the undamaged Sprint network description and traffic matrix. In almost every aspect, it is the same as the `attwdmg` module. The differences from `attwdmg` are described here, referencing the `attwdmg` module documentation where appropriate.

For `attwdmg`, it was assumed that backbone traffic was distributed in proportion to the trunks. In Sprint, this assumption does not hold. Therefore, `sprwdmg` requires an additional input file, which the `sprwdmg` code refers to as the morph file, to describe the undamaged traffic distribution. Refer to `sprlive` module documentation for more information on the calculation of this traffic distribution file.

The morph file is applied in the calculation of `SS`. Where `attwdmg` uses `IMT[ i][ j] /SUM_IMT[ i]` to distribute switch i's traffic to all other switches j, `sprwdmg` replaces this term with `Morph[ i][ j] /MorphRow[ i]`. In addition, the `SS` calculation for the Late case determines whether a connection exists between a pair of switches. The calculation for `sprwdmg` replaces the term `SUM_IMT[ i] ==0` with the term `MorphRow[ i] ==0` in the "if" clauses that determines connectivity between switches.

In all other respects, `sprwdmg` operates identically to `attwdmg`.

Inputs

character	`keyfile[ ]`		string containing the name of the key file

Outputs

global	`character[ 80]`	`eodmg_file`	the name of the EO damage file
		`atdmg_file`	the name of the AT damage file
		`eoatdmg_file`	the name of the EO-AT span damage file
		`regovl_file`	the name of the regional overload file
		`in_file`	the name of the input network description created by attlive
		`swdmg_file`	the name of the IEC switch damage file
		`popdmg_file`	the name of the IEC POP damage file
		`eopopdmg_file`	the name of the EO-POP span damage file
		`atpopdmg_file`	the name of the AT-POP span damage file
		`popswdmg_file`	the name of the POP-switch span damage file
		`qlink_file`	the name of the qlink file
		`morph_file`	the name of the IEC traffic distribution file

returns no formal values are returned

Purpose To read in the names of all necessary input files used by the main module

Called By `main()`

Calls To none

Local Variables

character	`line[ 80]`	temporarily holds a line of input from the keyfile
integer	`att, mci`	toggles used to indicate the presence of each IEC's data files in
	`spr`	the keyfile

Global Variables see Outputs above for others

file	`*fptr`	general file pointer variable used to open keyfile

Algorithmic Description

This function opens and processes the TAMI keyfile, which contains the list of all input files required to perform a TAMI run. Refer to Appendix B (TAMI Keyfile description) for more detail on the keyfile.

LoadKeyfile processes the TAMI keyfile using the following four steps:

1. Skip past comment lines (lines starting with '#')
2. Process IEC string ('ATT/MCI/SPR') to make sure Sprint is included
3. Skip to LEC file section and read LEC damage file names
4. Skip to SPR section and read Sprint input file names

Inputs none

Outputs none

returns no formal values are returned

Purpose To initialize all of the network and traffic variables before they are read in from the input files

Called By `main()`

Calls To none

Local Variables

integer	`i, j`	loop variables

Global Variables All global network and traffic variables. See main module description.

Algorithmic Description This function initializes all network and traffic variables. Traffic and trunk size quantities are initialized to 0. Damage vectors are initialized to 1 (survival). `Popswitch[ ]` is initialized to -1 to indicate no switch homed to each POP.

Inputs

file	`*inptr`	pointer to input IEC network file

Outputs Global network and traffic variables—described below under Algorithmic Description

returns no formal values are returned

Purpose To read in the undamaged network and traffic variables from the sprlive output file

Called By `main()`

Calls To none

Local Variables

integer	`i,j,k,m,n,L,P,S`	loop variables
	`offset, count`	record position and counting variables
character	`line[ 100] , temp[ 15]`	buffer memory

Global Variables Global network and traffic variables—described below under Algorithmic Description.

character	`template[ NUM_TEMP][ 15]`
constant	`NUM_TEMP`

Algorithmic Description

This function opens the undamaged network description created by sprlive. It reads in each section of the file, as indicated by the section labels in the `template[ ]` array. The following global variables are loaded from the input file:

NUMSW	AC
NUMPOP	ACSlec/ACSlec_live
NUMAT	TAP
NUMEO	SAP
ATPOPSIZE	D3/D3live
AT	ACPiec
HUPOP	popswitch
ATPOP	LATA
HAP	PhysDiv
SEP	IMT2
TEP	PropIntra
TEA	
FO	

Note that each of the 25 sections in the file may be stored in a different format and may have a different number of records depending on the variable type being read. For this reason, each section must be read in using the same formats used in sprlive when the file was written out. The dimensions, `NUMSW, NUMPOP, NUMAT, NUMEO`, and `ATPOPSIZE`, are also used to determine the sizes of the sections that follow.

Inputs

character	`*infile`	input file

Outputs

returns no formal returns

Globals:

real	`Rmult`	regional traffic multiplier
	`Lmult`	LATA traffic multiplier
integer	`LReg`	LATA to region mapping
	`EOReg`	EO to region mapping

Purpose This routine loads regional information.

Called By `main()`

Calls To library functions
`print_info(void)`

Local Variables

double	`multiplier`	
character	`buffer1[80], buffer2[80]`	working memory
integer	`i,lata, region`	loop variable
FILE	`*inputfile`	input file

Global Variables

integer
`NUMREGIONS`
`NUMEO`
`EOlata`
`EO_SIZE`

Algorithmic Description `Loadoverloads()` first opens the indicated file. For each region, it then retrieves the region designation and multiplier. It then reads one line for each LATA in the region. The overload and region designation for the lata is then set. `Loadoverloads()` then closes the file, and sets the region for each EO.

Inputs none

Outputs

returns no formal returns

Globals:

real `RmultMatrix`

Purpose This routine generates a matrix of region to region traffic multipliers..

Called By `main()`

Calls To library functions

Local Variables

integer	`i, j`	loop variable

Global Variables

real `Rmult`

Algorithmic Description This routine builds a symmetric region to region traffic multiplier matrix. The region to region traffic multiplier is taken as the max of the two regional multipliers.

Inputs none

Outputs

global	float	`ACSRegion[ NUM_SWITCH][ NUM_REGION]`

returns no formal values are returned

Purpose To categorize switch access traffic by region of origin for a regional focused overload run

Called By `main()`

Calls To none

Local Variables

integer	`k`	loop variable

Global Variables

integer	`EOReg`	for every EO, indicates the region it is in
	`SW`	for every EO, indicates its unique homed IEC switch
float	`F0dmg`	post-damage engineered access traffic at an EO

Algorithmic Description

This function sums up F0dmg for each EO to its home switch based on the region, EOReg, in which the EO is located. Its purpose is to aggregate and track, at the IEC switch level, the originating region of all access traffic. This result is important for calculating the regional SS matrix in function `MakeRegionalSS()`.

Inputs none

Outputs

global	float	AvgSwRmult[NUM_SWITCH][NUM_REGION]
	float	SSreg[NUM_SWITCH][NUM_SWITCH]
returns	no formal values are returned	

Purpose To compute the affect of regional focused overload on the backbone traffic distribution, and to measure the traffic-weighted overload created by regional affects for each originating switch/region combination.

Called By main()

Calls To none

Local Variables

integer	i,j	loop variables on dimension NUMSW
	Ri, Rj	loop variables on dimension NUMREGION

Global Variables

integer	EOReg	for every EO, indicates the region it is in
	SW	for every EO, indicates its unique homed IEC switch
float	ACSlec_dam	for every switch, indicates post-damage access traffic before regional overload is assessed
	RmultMatrix	regional overload affect for every pair of originating and terminating regions
	ACSRegion	access traffic offered from each region through each switch
	SS	non-regional backbone traffic distribution

Algorithmic Description

This function calculates the regional focused overload affect for each switch pair and stores the result in SSreg. For every switch pair, traffic is examined for every possible pair of regions between the switches. Each element of this traffic is subjected to overload defined by RmultMatrix. For example, from region Ri at switch i, to region Rj at switch j, traffic is overloaded by RmultMatrix[Ri][Rj] . These elements are weighted base on traffic at both ends, summed for all region combinations, and normalized to determine the average overload between the switch pair. This quantity, SStemp/ACSlec_dam, is multiplied by the original SS value to yield a regional overloaded SSreg value.

Note that an important intermediate result, AvgSwRmult, is calculated along the way. This value is the average focused-overload-induced multiplier applied to traffic leaving region Ri through switch i. It is computed from the overload experienced from region Ri at switch i to every other region/switch combination, Rj and j. This result has intentionally not been weighted by access traffic. This allows it to be used on a case-by-case basis for various access traffic values, as seen in function MakeRegionalF0dmg() .

Inputs none

Outputs

global	float	RegF0dmg[NUM_EO]

returns no formal values are returned

Purpose To compute the affect of regional focused overload on the backbone traffic distribution, and to measure the traffic-weighted overload created by regional affects for each originating switch/region combination.

Called By main()

Calls To none

Local Variables

integer	k	loop variable on dimension NUMEO

Global Variables

integer	EOReg	for every EO, indicates the region it is in
	SW	for every EO, indicates its unique homed IEC switch
float	F0dmg	for every EO, indicates the engineered access traffic prior to regional focused overload affects
	AvgSwRmult	indicates the traffic-weighted average regional focused overload experienced by access traffic from each region through each switch

Algorithmic Description This function calculates RegF0dmg for every EO. This quantity is a function of the post-damage engineered access traffic at the EO, F0dmg, and the average overload caused by regional overload effects, AvgSwRmult. AvgSwRmult, in turn, is a function of what region the EO resides in, and what switch it is homed to. For a regional focused overload, RegF0dmg represents the load offered at the EOs to the IEC.

Inputs

file	`*outptr`	pointer to output damaged IEC network file

Outputs Global network and traffic variables are written to file—described below under Algorithmic Description

returns no formal values are returned

Purpose To write out all the damaged network and traffic variables to the output file

Called By `main()`

Calls To none

Local Variables

integer	`k,l,m,n`	loop variables

Global Variables Global network and traffic variables—described below under Algorithmic Description.

Algorithmic Description This function writes a carrier-specific header to the output file, followed by the global damaged network and traffic variables. Each variable is written to a labeled section, with the following description:

`IECid`	identifies the carrier (att, mci, or spr)
`SWITCH_SIZE`	number of IEC switches in the network
`POP_SIZE`	number of POPs in the network
`AT_SIZE`	number of ATs in the network
`EO_SIZE`	number of EOs in the network
`ATPOPSIZE`	maximum number of POPs an AT homes to for this network
`EOAT`	for every EO, identifies the homed AT (-1 if N/A)
`HUPOP`	for every EO, identifies the homed HU POP
`ATPOP`	for every EO, identifies the homed POP through the AT
`HAP`	for every AT, identifies all homed POPs
`SD`	for every EO, identifies the size of the EO-POP TG
`TEA`	for every EO, identifies the total (both ways) traffic engineered for the EO-AT TG
`F0`	for every EO, identifies the components of access traffic offered to the HU and final homings respectively
`F0dmg`	for every EO, identifies the amount of engineered access traffic offered by the EO after damage affects are considered
`ELAP`	for every AT, identifies the total AT-POP traffic for each AT-POP homing defined in HAP
`SAP`	for every AT, identifies the size of each AT-POP TG
`ACS`	for every IEC switch, identifies the total access traffic offered for inter-switch transport
`POPSWITCH`	for every POP, identifies the homed IEC switch
`AC1`	for every EO, identifies the actual traffic offered, including regional focused overload, if applicable

EOalive	for every EO, indicates 0/1 for failure/survival
ATalive	for every AT, indicates 0/1 for failure/survival
POPalive	for every POP, indicates 0/1 for failure/survival
SWalive	for every switch, indicates 0/1 for failure/survival
EOlata	for every EO, identifies its LATA
PhysDiv	for every EO, identifies if the HU homing (if any) is physically diverse from the final homing (if any)
SS	for the matrix of IEC switch pairs, identifies traffic distribution from one switch to the other
SSreg	same as SS, but for regional focused overload
PropIntra	for every switch, indicates the proportion of access traffic that is intra-switch traffic

Note that each of the sections in the file may be stored in a different format and may have a different number of records depending on the variable type being written.

3.10 merge: Merge of IEC Files

Purpose This is the main module for merging the iec2lecam.step2 files

Call Syntax `mkrout -k <key file>  [ options]`

mandatory:	*special syntax:*	*function*
none		
options:	*special syntax:*	*function:*
-k	<input key file>	reads in input file
-?		user help—prints call syntax and exits without running

example `merge -k  keyfile`

Input Files The files listed below are input through a command line option.

`IEC_SIZE`	IEC data file

Output Files

`outfile`	This file contains the results of the combined `.step2` files

Includes

`"fileio.c"`	User-defined I/O functions; see Appendix A
`"lecam.h"`	User-defined I/O functions

Constants

A005	211.10	traffic reqd for P.005 on 250 trunks
A01	214.60	traffic reqd for P.01 on 250 trunks

Global Variables

integer

KEYTOG=0	keyfile toggle
IEC_SIZE	size of the IEC
SWITCH_SIZE[MAX_IEC]	size of the IEC switch
POP_SIZE[MAX_IEC]	size of the POP
AT_SIZE	size of the AT
EO_SIZE	size of the EO
ATPOPSIZE[MAX_IEC]	size of the AT to POP connection
EOAT[MAX_EO]	size of the EO to AT connection
HUPOP[MAX_EO] [MAX_IEC]	amount of traffic on the HU to POP connection
ATPOP[MAX_EO] [MAX_IEC]	amount of traffic on the AT to POP connection
HAP[MAX_IEC] [MAX_AT] [ATPOPMAX]	amount of traffic on the HU-AT to POP connection
SD[MAX_EO] [MAX_IEC]	number of trunks EO to AT
SAP[MAX_IEC] [MAX_AT] [ATPOPMAX]	number of trunks AT to POP
POPSWITCH[MAX_IEC] [MAX_POP]	locatioon of the POP switch
EOalive[MAX_EO]	surviving EO
ATalive[MAX_AT]	surviving AT
POPalive[MAX_IEC] [MAX_POP]	surviving POP

	`SWalive[ MAX_IEC][ MAX_SWITCH]`	surviving Switch
	`EOlata[ MAX_EO]`	LATA associated with each EO
	`PhysDiv[ MAX_EO][ MAX_IEC]`	co-location indicator
real	`SS`	the engineered prop of traffic from SW S1 to each SW S2 norm by row
	`Ssreg`	the proportion of traffic from SW S1 to each SW S2 normalized by row
kc2type	`F0`	total access traffic from EO k to IEC c ([0]HU [1]FINAL)
kctype	`F0dmg`	access traffic with damage from EO k to IEC
	`AC1`	access traffic at EO k destined for IEC c
cLPtype	`ELAP`	engineered load on AT-POP TG for AT L and POP P in IEC c
cStype	`ACS`	engineered level of access traffic at SWITCH S
character	`template[ 27][ 15] =`	{"IECid","SWITCH_SIZE","POP_SIZE","AT_SIZE", "EO_SIZE","ATPOPSIZE","EOAT","HUPOP","AT POP","HAP","SD","TEA","F0","F0dmg","ELAP","SAP","ACS","POPSWITCH","AC1","EOalive","ATalive","POPalive","SWalive","EOlata","PhysDiv","SS","SSreg"};
	`inputfile[ MAX_IEC][ 80]`	name of the input file
	`outputfile[ MAX_IEC][ 80]`	name of the output file
	`qfile[ MAX_IEC][ 80]`	name of the QTCM file
	`IECid[ MAX_IEC][ 5]`	IEC inidcator (att/mci/spr)

Local Variables

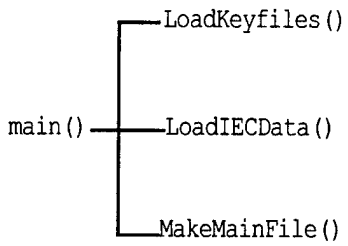
Variables local to `main()`

extern	character	`*optarg`	a string containing a single command line argument.
	integer	`optind`	the number of single command line arguments to be processed, supplied externally by the operating system
character	`c`		a command line option character
	`filelist[ 80]`		holds the name of the input key file
	`**argv[ ]`		an array containing all of the command line arguments
integer	`tog=0`		error flag indicating a problem with the command line arguments
	`argc`		the number of command line arguments

Component Functions

`LoadKeyFiles()`	loads input key files
`LoadIECData()`	loads IEC data
`MakeMainFile()`	formats and writes output file

Function Tree



Algorithmic Description

This module is used to merge the three iec `.step2` files:

- `att2lecam.step2`
- `mci2lecam.step2`
- `spr2lecam.step2`

Inputs

character	`*keyfile`	pointer to an input file

Outputs

returns	no formal returns	
FILE	`*inputfile`	points to the input file containing the `TAMI.iec2lecam.step2` data
	`*qfile`	points to the `TAMI.qtcm.iec` qtcm data file
	`*outputfile`	points to the `TAMI.dp.iec` output file

Purpose This routine loads

Called By `main()`

Calls To none

Local Variables

character	`line[ 80]`	used for parsing the input file
integer	`i`	loop count variable
	`count=0`	loop count variable
	`att=0`	iec toggle
	`mci=0`	iec toggle
	`spr=0`	iec toggle
FILE	`*fptr`	pointer to an input file

Global Variables none

Algorithmic Description For each iec, this routine loads the names of the input files into the appropriate input, output and qtcm files.

Inputs

integer	c	loop count variables
FILE	*fptr	points to an output file

Outputs

integer	KEYTOG=0	keyfile toggle
	IEC_SIZE	size of the IEC
	SWITCH_SIZE[MAX_IEC]	size of the IEC switch
	POP_SIZE[MAX_IEC]	size of the POP
	AT_SIZE	size of the AT
	EO_SIZE	size of the EO
	ATPOPSIZE[MAX_IEC]	size of the AT to POP connection
	EOAT[MAX_EO]	size of the EO to AT connection
	HUPOP[MAX_EO][MAX_IEC]	amount of traffic on the HU to POP connection
	ATPOP[MAX_EO][MAX_IEC]	amount of traffic on the AT to POP connection
	HAP[MAX_IEC][MAX_AT][ATPOPMAX]	amount of traffic on the HU-AT to POP connection
	SD[MAX_EO][MAX_IEC]	
	SEA[MAX_EO]	number of trunks EO to AT
	SAP[MAX_IEC][MAX_AT][ATPOPMAX]	number of trunks AT to POP
	POPSWITCH[MAX_IEC][MAX_POP]	locatioon of the POP switch
	EOalive[MAX_EO]	surviving EO
	ATalive[MAX_AT]	surviving AT
	POPalive[MAX_IEC][MAX_POP]	surviving POP
	SWalive[MAX_IEC][MAX_SWITCH]	surviving Switch
	EOlata[MAX_EO]	LATA associated with each EO
	PhysDiv[MAX_EO][MAX_IEC]	co-location indicator
real	SS	the engineered prop of traffic from SW S1 to each SW S2 norm by row
	Ssreg	the proportion of traffic from SW S1 to each SW S2 normalized by row
kc2type	F0	total access traffic from EO k to IEC c ([0]HU [1]FINAL)
kctype	F0dmg	access traffic with damage from EO k to IEC
	AC1	access traffic at EO k destined for IEC c
cLPtype	ELAP	engineered load on AT-POP TG for AT L and POP P in IEC c
cStype	ACS	engineered level of access traffic at SWITCH S
character	template	IEC data template

	`inputfile`	holds the name of the input file
	`outputfile`	holds the name of the output file
	`qfile`	holds the name of the QTCM file
	`IECid`	identified the IEC

returns no formal returns

Purpose This routine loads the IEC data for each IEC

Called By `main()`

Calls To none

Local Variables

integer	`i,j,k,L,P,S`	loop count variables
	`offset`	used to parse a line of the input file
	`count`	loop count variable
FILE	`*outptr`	points to an output file
character	`line[ 100]`	used to hold a line of input from the input file
	`temp[ 10]`	used to hold a line of input from the input file

Global Variables

character	`line[ LINE_LENGTH]`	length of input line

Global Constants `none`

Algorithmic Description Reads the IEC data fields listed as outputs and loads them into the output file.

Inputs

FILE	*fptr	pointer to the output file

Outputs

returns no formal returns

Purpose To produce the output file.

Called By `main()`

Calls To none

Local Variables

integer	`k,c,L,P,S`	loop count variables
	`count`	loop count variable
real	`x`	used to track traffic EO to AT

Global Variables

integer	`num_11=0`	number of class1 to class1 switch pairs
	`num_13=0`	number of class1 to class3 switch pairs
	`num_33=0`	number of class3 to class3 switch pairs
character	`line[ LINE_LENGTH]`	length of input line

Algorithmic Description This routine prints out the output data file for the datafields listed as Global Variables for this module.

3.11 LECAM: Determines Overall PSN Blockage

Purpose

LECAM is the module of TAMI which offers traffic at the EO level and calculates subsequent blockages in every TG in the PSN. This module contains the code which loads the IEC and LEC datafiles (including network topology, physical damage, and offered traffic) and performs LEC blockage calculations. For IEC blockages, LECAM calls QTCM for each IEC. The code which performs the summary calculations for overall PSN blockage is contained in `calcbigb.c`.

Call Syntax

`lecam -i <input file> [ options]`

mandatory:	*special syntax:*	*function:*
-i	<input file>	reads in input file

options:	*special syntax:*	*function:*
-d		DOC ON/OFF
-p		use Poisson blockage ON/OFF
-f		flush screen buffer (for debug)
-o	<outfile>	output logfile
-c	<filename>	CSI link file
-m	<float>	access traffic multiplier
-b	<integer>	initial number of pairs sampled
-s	<integer>	random number stream (1-15)
-e	<1|2>	turn on early-post-disaster rules
-C		enable count of ALL connected pairs through all IECs
-v		init assumed variables with final values from prev run
-n	<filename>	Use NS/EP traffic matrix nij_file
-r	<filename>	Use Regional
-A		Turn Glare Off
-B	<float>	Set DOC Level
-D		Turn Matching Loss Off
-E		Use High Matching Loss Levels
-F	<float>	Set Matching Loss Level
-x		cancel PrintResults
-l		use LEC TCR in select LATAs
-?		help

example `lecam -i TAMI.dp.main -l -m 1.25 -d -el -n traffic_mat`

Input Files

regional overloads — This file contains region definitions and traffic overloads

format

`R<#> <overload value>`
The region number, and traffic overload value
`<LATA #>`
`[ <LATA #>`
`...]`
A list of LATAs in the region

```
[ R<#> <overload value>
<LATA #>
[ <LATA #>
...]
...]
```
A list of region definitions, as above.

main datafile Contains LEC assets and sizings, damage vectors, and LATA definitions. It also contains the names of the IEC input data files, and the total number of IECs under consideration.

format

`IEC_SIZE <#>`
Beginning in column 13, the number of IECs under consideration.

`IECfiles`
`<3-character IEC id> <IEC data file>`
Beginning in column two, the three character IEC identifier. Then at column five, the filename (pathname) of the datafile for the referenced IEC.

`AT_SIZE <#>`
Beginning in column 13, the number of ATs in the model.

`EO_SIZE <#>`
Beginning in column 13, the number of EOs in the model.

`EOAT`
`<4-character AT index><4-character AT index>...`
For each EO, a four column identifier for its homed AT. These four character indexes are arranged twenty to a line.

`SEA`
`<5-character size><5-character size>...`
For each EO, a five character size of the EO-AT trunk. These blocks are arranged 15 to a line.

`EOalive`
`<1|0><1|0><1|0><1|0>...`
For each EO, a single character (1 or 0) indicating whether that EO is to be considered alive in this run. Arranged 80 to a line.

`ATalive`
`<1|0><1|0><1|0><1|0>...`
For each AT, a single character (1 or 0) indicating whether that AT is to be considered alive in this run. Arranged 80 to a line.

`EOlata`
`<4-character LATA #><4-character LATA #>...`
For each EO, a four column identifier for its LATA. These four character identifiers are arranged twenty to a line.

IEC datafiles Contains IEC assets and sizings, damage vectors, and default traffic distributions.

format

`SWITCH_SIZE <#>`
Beginning in column 13, the number of switches belonging to this IEC which are in the model.

`POP_SIZE <#>`

Beginning in column 13, the number of POPs in the model.

AT_SIZE <#>
Beginning in column 13, the number of ATs in the model.

EO_SIZE <#>
Beginning in column 13, the number of EOs in the model.

ATPOPSIZE <#>
Beginning in column 13, the number of AT-POP trunks belonging to this IEC which are in the model.

QTCMfile <filename>
Beginning in column nine, the filename (pathname) of the QTCM datafile for the referenced IEC.

HUPOP
<4-character POP index><4-character POP index>...
For each EO, a four column identifier for any direct POP HU trunk. These four character indexes are arranged twenty to a line.

ATPOP
<4-character POP index><4-character POP index>...
For each EO, a four column identifier for any AT-POP trunk. These four character indexes are arranged twenty to a line.

HAP
<4-character POP code><4-character POP code>...
<4-character POP code><4-character POP code>...
<4-character POP code><4-character POP code>...
...
For each AT, and each POP for the current IEC, a four character code indicating whether the AT is homed to that POP.

SD
<5-character size><5-character size>...
For each EO, a five character size of the EO-POP HU trunk. These blocks are arranged 15 to a line.

F0
<8-character traffic value 0><8-character traffic value 1>
<8-character traffic value 0><8-character traffic value 1>
...
For each EO, a line with two eight character traffic values.

F0dmg
<8-character traffic value>
<8-character traffic value>
...
For each EO, a line with an eight character traffic values.

ELAP
<8-character traffic value><8-character traffic value >...
<8-character traffic value><8-character traffic value >...

```
<8-character traffic value><8-character traffic
value >...
...
```
For each AT, and each POP for the current IEC, an eight character engineered access traffic load value.

```
SAP
<4-character trunk size><4-character trunk size
>...
<4-character trunk size ><4-character trunk size
>...
<4-character trunk size ><4-character trunk size
>...
...
```
For each AT, and each POP for the current IEC, an eight character trunk size.

```
ACS
<8-character traffic value><8-character traffic
value >...
```
For each switch an eight character engineered access traffic load value. These are arranged ten to a line.

```
POPSWITCH
<4-character switch code><4-character switch
code>...
<4-character switch code><4-character switch
code>...
<4-character switch code><4-character switch
code>...
...
```
For each POP, and each switch for the current IEC, a four character code indicating whether the POP is homed to that switch.

```
AC1
<8-character traffic value><8-character traffic
value >...
```
For each EO an eight character engineered access traffic load value. These are arranged ten to a line.

```
POPalive
<1|0><1|0><1|0><1|0>...
```
For each POP in the current IEC, a single character (1 or 0) indicating whether that POP is to be considered alive in this run. Arranged 80 to a line.

```
SWalive
<1|0><1|0><1|0><1|0>...
```
For each switch in the current IEC, a single character (1 or 0) indicating whether that switch is to be considered alive in this run. Arranged 80 to a line.

```
PhysDiv
<1|0><1|0><1|0><1|0>...
```
For each EO, a single character (1 or 0) indicating whether that EO's HU is physically diverse from its AT-POP final. Arranged 80 to a line.

SS
<8-character traffic value><8-character traffic value >...
<8-character traffic value><8-character traffic value >...
<8-character traffic value><8-character traffic value >...
A square switch-switch matrix of eight character values indicating the proportion of traffic destined from one switch to the other.

SSregress
<8-character traffic value><8-character traffic value >...
<8-character traffic value><8-character traffic value >...
<8-character traffic value><8-character traffic value >...
...
A square switch-switch matrix of eight character values indicating the proportion of traffic destined from one switch to the other. This is normalized by column.

TAMI.finalvalues

Carryover results from the previous iteration.

format

OLAT
<OLAT value> <OLAT value> <OLAT value>...
A value for each AT traffic overload.

OLEO
<OLEO value> <OLEO value> <OLEO value>...
A value for each EO traffic overload.

For each IEC:
e2 and SWBlock
<e2 value> <e2 value> <e2 value>...
An egress blockage value for each AT-POP trunk.

<SWBlock value><SWBlock value>...
Switch to switch blocking values.

traffic sample file

EO-EO traffic values for randomly selected EO pairs.

format

<EO index #> <EO index #> <traffic value>
The first two integers index EOs. Beginning at column 12, a floating point number representing the offered traffic between that pair.

ppmeanfile.<att|mci|spr>

Average IEC blockage, and switch to switch IEC blockage.

format

<average blockage value>

<7-character traffic value><7-character traffic value>...
<7-character traffic value><7-character traffic value>...

```
<7-character traffic value><7-character traffic
value>...
```

A square switch-switch matrix of seven character values indicating the blockage from one switch to the other.

Traffic Matrix

EO-EO pair traffic values

format

```
<EO 1> <EO 2> <traffic value>
...
```

A listing of EO pairs, followed by the traffic between them.

Output Files

TAMI.finalvalues

As documented above.

TAMI.lecam.qtcm.temp

Documented as a QTCM input file.

screen output (captured to `TAMI.latest_lecam_outfile` when invoked from tami.c)

This basic file structure may be interspersed with warning messages printed by LECAM.

format

```
DOC = <#>
[ DOC invoked at = <#>]
<EARLY|LATE> POST-DISASTER MODE
Traffic Multiplier: <#> times.
<#> outer loops on SWBlock (QTCM).
<#> inner loops on e2 (on average).
<#> innermost loops on EO and AT overload (on
average).
```

Fairly self-explanatory settings and convergence reporting.

```
OLEO and Blockage on EO TGs:
EO <EO index> (<0=dead|1=alive>): OL=<#> (a1=<#>,
e1=<#>)
...
```

For each EO, its index, alive state, overload and blockages.

```
EO <EO index> (<0=dead|1=alive>): OL=<#> (a1=<#>,
e1=<#>)
(a3=<#>).(a0=<#>, e0=<#>)...
```

For each EO, its index, alive state, overload and blockages.

```
OLAT and Blockage on AT-IEC TGs (a2,e2)
AT <AT index> (<0=dead|1=alive>) OL=<#>
IEC<IEC id #>:  a2=<#> e2=<#>
...
```

For each AT, its index, alive state, overload and blockages broken out by carrier.

TAMI.lecamlog

Keeps a running log of LECAM results.

format

```
<EARLY|LATE> POST-DISASTER MODE
Traffic Multiplier[ s] : <<#> times | [ R<#>=<#>]
[ R<#>=<#>]
...>
```

Traffic overloading. If a regional analysis, several lines of region/overload pairs.

```
[ DOCLEVEL: #]
```

If DOC has been invoked, the traffic level (multiple of engineered load) at which it becomes active.

```
[*** OSCILLATION DETECTED:  <#> <OL|e2|SWBlock>
variables] ...
```

Warns of oscillations in the three variable classes for which this is tested, and gives the number of variables oscillating.

```
Survival data:
  EOs..... <%>
  ATs..... <%>
  POPs..... [ <%> (IEC <IEC id #>)]
[ <%> (<IEC id #>)]
...
  Switches.... [ <%> (IEC <IEC id #>)]
[ <%> (<IEC id #>)]
...
  EO-POP TGs.... [ <%> (IEC <IEC id #>)]
[ <%> (<IEC id #>)]
...
  AT-POP TGs.... [ <%> (IEC <IEC id #>)]
[ <%> (<IEC id #>)]
...
  EO-AT TGs..... <%>
```

The percentage of surviving assets of the indicated type.

```
POTS Traffic Totals for IEC <IEC id #> (<IEC
id>):
     DOC engineered=   <#>
  NO DOC engineered=   <#>
      offered to EO=   <#>
  offered after DOC=   <#>   (<%>)   (<%>)
     offered to IEC=   <#>   (<%>)   (<%>)
 completed thru IEC=   <#>   (<%>)   (<%>)
    completed to EO=   <#>   (<%>)   (<%>)
    percent blocked=   <#>   (<%>)   (<%>)
```

For each IEC, cumulative statistics on traffic blockage across each major PSN component.

```
TOTAL ENGINEERED TRAFFIC: <#> erlangs (<%>%,
[ <%>%, ] ...)
TOTAL OFFERED TRAFFIC:    <#> erlangs (<%>%,
[ <%>%, ] ....)
TOTAL COMPLETED TRAFFIC:  <#> erlangs (<%>%,
[ <%>%, ] ...)
```

Traffic totals for each IEC.

```
Connectivity Analysis:
PSN     : POTS=<#>
  NSEP/RTNR=<#>
```

```
  [ NSEP/CSI=<#>]
IEC <#> (<IEC id>): POTS=<#>
  NSEP/RTNR=<#>
  [ NSEP/CSI=<#>]
...
```

The percentage connectivity for the PSN and each IEC.

```
BIGBPOTS=<#>
>BIGBPOTS<IEC id>=<#>
BIGBSAMPr<region1>r<region2>=<#>
>BIGBSAMPr<region1>r<region2><IECid>=<#>
...
```

Traffic-weighted blockage values for the PSN based on POTS traffic, and the provided traffic sample, respectively.

```
BIGBRTNR<0|1>Or<region1>r<region2>=<#>
>BIGBRTNROr<region1>r<region2><IECid>=<#>
...
BIGBCSI<0|1>r<region1>r<region2>=<#>
>BIGBCSIr<region1>r<region2><IECid>=<#>
...
BIGBTCR<0|1>r<region1>r<region2>=<#>
>BIGBTCR0r<region1>r<region2><IECid>=<#>
...
BIGBLEC<0|1>r<region1>r<region2>=<#>
>BIGBLECOr<region1>r<region2><IECid>=<#>
...
```

The blockages associated with the addition of each indicated program (RTNR, CSI, IEC TCR, LEC TCR). The <0|1> indicates whether the traffic weighting used supplied sample traffic. One block is printed for each region-region pair.

```
BIGBCONNpsn=<#>
BIGBCONNpsnr=<#>
[ BIGBCONNpsn_csi=<#>]
```

The connectivity associated with the addition of each indicated program (RTNR, CSI).

```
BIGBCONN<IEC id>=<#>
[ BIGBCONNrtnr=<#>]
[ BIGBCONNcsi=<#>]
```

The connectivity associated with the addition of each indicated program (RTNR, CSI).

Includes

`<stdio.h>`	Standard 'c' defined I/O functions.
`<math.h>`	Standard 'c' defined math functions.
`"fileio.c"`	User-defined I/O functions; see Appendix A.
`"lecam.h"`	User-defined typedefs and defines.

Constants

`TEST_SW_R1`	debugging switch
`TEST_EO_R1`	debugging switch
`TEST_SW_R0`	debugging switch
`TEST_EO_R0`	debugging switch

from `lecam.h`

`TRUE`	1	true
`FALSE`	0	false

MAX_IEC	3	max number of IECs in model
MAX_AT	1000	max number of ATs in model
MAX_EO	19250	max number of EOs in model
ATPOPMAX	9	max number of POPs a single AT may home to
MAX_POP	575	max number of POPs in an IEC
MAX_SWITCH	130	max number of switches in an IEC
MAX_LATA	1000	max number of LATAs in model
MAX_REGION	5	max number of regions in model

Data Types

from `lecam.h`

These pervasive type definitions are named very regularly. The indexes used are stated in the type name, in order:

```
c  carrier
k  EO
L  AT
P  POP
R  Regression
S  Switch
```

```
typedef float    ctype[ MAX_IEC]
typedef float    ktype[ MAX_EO]
typedef float    Ltype[ MAX_AT]
typedef float    kctype[ MAX_EO][ MAX_IEC]
typedef float    kc2type[ MAX_EO][ MAX_IEC][ 2]
typedef float    cLPtype[ MAX_IEC][ MAX_AT][ ATPOPMAX]
typedef float    cPtype[ MAX_IEC][ MAX_POP]
typedef float    cStype[ MAX_IEC][ MAX_SWITCH]
typedef float    cSRtype[ MAX_IEC][ MAX_SWITCH][ MAX_REGION]
typedef float    cSStype[ MAX_IEC][ MAX_SWITCH][ MAX_SWITCH]
typedef float    cPPtype[ MAX_IEC][ MAX_POP][ MAX_POP]
typedef float    RRctype[ MAX_REGION][ MAX_REGION][ MAX_IEC]
```

Global Variables

/***** Default Command Line Toggles ********/**

integer	DOC	0	DOC ON (1), OFF (0) -- Default is OFF
	STOREDOC	0	DOC toggle setting from command line (same as DOC until NSEP traffic)
	POI	0	trunk blockage function: Poisson (1), Erlang B (0) default is Erlang B
	BUNDLE_SIZE	100	initial sample target
	BUNDLE_INC	20	incremental number of samples
	EARLY	0	toggle for early-post disaster (0 LATE, 1 EARLY OPT 1, 2 EARLY OPT 2)
	COUNTPAIRS	0	toggle to count ALL connected pairs through carriers default is OFF
	CSI	0	toggle CSI -- default is OFF (CSI filename is passed with -c option)
	RSTREAM	1	random number stream (1-15) default is 1
	INITPREV	0	initialize assumed variables with final values from previous (TAMI.finalvalues)
	NSEPMAT	0	toggle to use NS/EP traffic matrix default is OFF
	RTOG	0	toggle to use REGIONAL FOCUSED OVERLOAD default is OFF

	CANCELPR	0	cancel `PrintResults` (`LogResults` still OK)
	LECTCR	0	toggle to use TCR in selected LATAs
real	Tmult	1.0	input traffic multiplier default 1.0 times

/******* Sensitivity Test Toggles ********/**

integer	GLAREOFF	0	(-A) Turn glare equations OFF (1) or ON (0)
	SETDOC	0	(-B) Specify DOC level ON (1) or OFF (0) -- held in SETDOCLEVEL
	MATCHOFF	0	(-D) Turn matching loss OFF (1) or ON (0)
	HIGHMATCH	0	(-E) Use high matching loss levels ON (1) or OFF (0)
	SETMATCH	0	(-F) Specifiy matching loss level ON (1) or OFF (0)
real	SETDOCLEVEL		level at which DOC is invoked
	SETMATCHLEVEL		(-F) Level at which matching loss begins
	DOCLEVELDOCDEFAULT		DOC overload level (default is DOCDEFAULT)

/***** Global Variables **********/**

ctype	b	average blockage thru IEC c over country
	CONNpots	average EO-to-EO POTS connectivity thru IEC c
	CONNcsi	average EO-to-EO NSEP w/ CSI connectivity thru IEC c
	CONNrtnr	average EO-to-EO NSEP w/ RTNR connectivity thru IEC c
RRctype	BigBpots	average POTS EO-to-EO blockage thru IEC c
	BigBcsi	average NS/EP w/ CSI EO-to-EO blockage thru IEC c
	BigBrtnr0	average NS/EP w/ RTNR (0=POTS weighting, 1=NSEP weighting)
	BigBrtnr1	
	BigBtcr0	average NS/EP w/ TCR (0=POTS weighting, 1=NSEP weighting)
	BigBtcr1	
	BigBlec0	average NS/EP w/ LEC TCR (0=POTS weighting, 1=NSEP weighting)
	BigBlec1	
real	CONNpsn_pots	sampled PSN connectivity
	CONNpsn_rtnr	
	CONNpsn_csi	
kctype	a0	access blockage on direct (HU) TG from EO k to IEC c
	e0	egress blockage on direct (HU) TG from IEC c to EO k
	a3	average access blockage due to network controls (DOC) */
	F0dmg	access traffic with damage from EO k to IEC c */
kc2type	F0	total access traffic from EO k to IEC c ([0]HU [1]FINAL) */
ktype	a1	access blockage on EO-AT TG from EO k */
	e1	egress blockage on EO-AT TG to EO k */
	F	total engineered access traffic from EO k */
	ELEA	engineered load on EO-AT TG for EO k
cLPtype	a2	access blockage on AT-IEC TG thru AT L to POP P in IEC c
	e2	egress blockage on AT-IEC TG from POP P in IEC c to AT L
	e2new	
	ELAP	engineered load on AT-POP TG for AT L and POP P in IEC c
cStype	ACS	engineered level of access traffic at SWITCH S

	ACSdmg	access traffic summed to switch S including Tij multiplier
	ACSdmg_nomult	access traffic summed to switch S excluding Tij multiplier
	ACSdmg_prime	sum of ACIprime (used in calct.c)
	ACSlec_live	sum of F0s
	ERF	egress Reduction Factor - holds the proportion of egress traffic a switch should receive to reduce overload
	OLSW	The overload at a switch
	SSnormal	the sum of the matrix columns in SSreg to normalize SSrev
	TRAF_KEEP	the amount of traffic kept at a switch after all other switches have been checked for DOC
	TRAF_DELT	
	DIFF	the amount of traffic that was clipped by exceeding engineered access traffic
Ltype	ATEL	engineered traffic at AT L (all trunks) loads
cSStype	SWBlock	individual SW-to-SW blockages for each IEC switch pair
	SWBlocknew	
	SS	the engineered prop of traffic from SW S1 to each SW S2 norm by row
	SSregress	the proportion of traffic from SW S1 to each SW S2 normalized by row
	SSTNG	
	SSregrev	the proportion of traffic from SW S1 to each SW S2 normalized by col
	ACSRdmg	access traffic summed to switch S from Regression R including Tij multiplier
integer	SEA[MAX_EO]	size of TG from EO k to its AT
	SAP[MAX_IEC] [MAX_AT] [ATPOPMAX]	size of TG from AT L to POP P in IEC c
	EOalive[MAX_EO]	EO damage vector: 1=alive, 0=dead
	ATalive[MAX_AT]	AT damage vector
	POPalive[MAX_IEC] [MAX_POP]	POP damage vectors
	SWalive[MAX_IEC] [MAX_SWITCH]	switch damage vectors
	EOlata[MAX_EO]	lata to which each EO is assigned
	PhysDiv[MAX_EO] [MAX_IEC]	phys diversity of an HU TG from a final TG: 1=YES, 0=NO
	POP_SIZE[MAX_IEC]	the actual number of POPs in IEC c
	SWITCH_SIZE[MAX_IEC]	the actual number of switches in IEC c
	EO_SIZE	the actual number of EOs
	IEC_SIZE	the actual number of IECs
	AT_SIZE	the actual number of ATs
	ATPOPSIZE[MAX_IEC]	the actual max number of POPs homed to an AT in IEC c

/* Regional Variable Definitions */

integer	NUMREGIONS 1	The number of regions being analyzed
real	Lmult[MAX_LATA]	Traffic multiplier for a lata
	Rmult[MAX_REGION]	Traffic multiplier for a region
	RmultMatrix[MAX_REGION] [MAX_REGION]	Traffic multiplier used for traffic going from region A to region B
integer	LRegress[MAX_LATA]	Region of a lata

	EORegress[MAX_EO]	Region of an EO

/* Access Traffic Variable Definitions */

kctype	AC1	access traffic at EO k destined for IEC c
	AC1prime	access traffic at EO k destined for IEC c after network controls
	AC3	access traffic delivered to POP over HU TG
	AC4	overflow from direct TG -- offered to EO-AT TG
	AC6	access traffic from EO k delivered to AT for IEC c
ktype	AC2	total access traffic at EO k
	AC5	total access traffic off'd EO-AT TG, used for blockage
Ltype	AC7	total access traffic off'd AT, used for matching loss
cLPtype	AC8	total access traffic off'd AT-IEC TG for IEC c
cPtype	AC9	access traffic delivered to IEC c from AT trunks
cSRtype	AC10	access traffic delivered to IEC c SWITCH S from REGION R
	AC13	access traffic delivered to IEC c backbone from REGION R
cStype	AC11	total access traffic delivered to IEC c
	AC12	total access traffic delivered to IEC c backbone from switch S
	AC14normal	used to renormalize the SSregress matrix
cSStype	AC14	
	AC14adj	the switch-to-switch matrix offered to QTCM
	AC14delt	the difference between AC14adj and AC14

/* Egress Traffic Variable Definitions */

cStype	EG1	egress traffic at IEC SWITCH
	EG1engineered	
	EG1final	
	EGtemp	
cSRtype	EG1R	egress traffic at SWITCH from REGION
	EG1Rw	Engineered egress traffic at SWITCH if all traffic came from REGION weighted by destination regions, used only as a proportion to calc EG2
kctype	EG2	egress traffic at IEC for EO k
	EG3	egress traffic delivered to EO k on HU TG
	EG4	egress traffic off'd to IEC-AT TG for EO k from IEC c
	EG6	egress traffic delivered to AT from IEC c for EO k
	EG9	egress traffic delivered to EO k from IEC c thru AT
	EG10	total egress traffic at EO k from IEC c
	EG12	total egress traffic complete thru EO k from IEC c
cLPtype	EG5	total egress traffic offered IEC-AT TG, used for blockage
Ltype	EG7	total egress traffic at AT, used for matching loss
ktype	EG8	total egress traffic off'd AT-EO TG for EO k
	EG11	total egress traffic at EO k, used for matching loss
Ltype	OLAT	traffic overload at each AT L
	OLATnew	
ktype	OLEO	traffic overload at each EO k
	OLEOnew	
ctype	offdtraf	
real	w, x, x1, y, z	
double	a, q	

integer	test, i, j, k, L, c, PM, P, P1, P2, S, R	
	bcount	loop counters
	Olcount	
	e2count	
	totalOLcount	loop counters
	totale2count	
	b_not_converge[2]	number of variables not converged from prev two iterations
	e2_not_converge[2]	
	ol_not_converge[4]	number of variables not converged from prev four iterations
	b_oscillation	zero if no oscillation detected, otherwise holds number of oscilation variables
	e2_oscillation	
	ol_oscillation	zero if no oscillation detected, otherwise holds number of oscilation variables
	FirstLoop	toggle to indicate the first iteration
	EOAT[MAX_EO]	the AT to which EO k is homed (-1 if none)
	HUPOP[MAX_EO][MAX_IEC]	the POP in IEC c to which EO k is homed for HU
	ATPOP[MAX_EO][MAX_IEC]	the POP in IEC c to which EO k is homed for final traffic (-1 if none)
	HAP[MAX_IEC][MAX_AT][ATPOPMAX]	the POP(s) in IEC c to which an AT L is homed (-1 if none)
	SD[MAX_EO][MAX_IEC]	the size of the HU TG from EO k to IEC c (0 if none)
	POPSWITCH[MAX_IEC][MAX_POP]	the switch to which POP P is homed for IEC c (-1 if none)
	EOSW[MAX_EO][MAX_IEC]	returns the homed IEC c switch given EO k
	attnum	
	mcinum	
	sprnum	
long integer	long_count	long integer counter for big loops
	long_count2	long integer counter for big loops
char	input_file[80]	main input data file
	nij_file[80]	Nij filename (NS/EP)
	IECfiles[MAX_IEC][80]	names of each IECs data files
	QTCMfile[MAX_IEC][80]	names of each IECs QTCM data file
	CSIfile[80]	names of CSI link file (optional)
	IECid[MAX_IEC][5]	three-character IEC identifier: att, mci, spr
	regressovl_file[80]	name of regional overload file (optional)
	MainTemplate[9][15]	{"IEC_SIZE", "IECfiles", "AT_SIZE", "EO_SIZE", "EOAT", "SEA", "EOalive", "ATalive", "EOlata"} order of variables in main data file
	SubTemplate[22][15]	{"SWITCH_SIZE", "POP_SIZE", "AT_SIZE", "EO_SIZE", "ATPOPSIZE", "QTCMfile", "HUPOP", "ATPOP", "HAP", "SD", "F0", "F0dmg", "ELAP", "SAP", "ACS", "POPSWITCH", "AC1", "POPalive", "SWalive", "PhysDiv", "SS", "SSreg"} order of var in IEC data files

Local Variables

Variables local to `main()`

integer	`err=0`	error flag indicating a problem with the command line arguments
	`argc`	the number of command line arguments
	`i, j`	loop counter variables
	`current_bin`	
	`vector_offset`	
	`OPTOG`	indicates command line options
FILE	`*fptr`	file pointer

Component Functions

```
void CalcAE(float b, float AC, float EG, float *a,
float *e)
void MakeQFile(integer c, char qtcm_file[])
float QResult(char fname[])

void ReadQMatrix(cSStype SWBlock,integer IEC, char
fname[])
LoadPrevValues(FILE *fptr)
void DumpFinalValues()
LoadData(char infile[])
LoadMainFile(FILE *fptr)
LoadSubFile(integer c, char filename[])
LogResults(FILE *fptr)
PrintResults()
void PrintBlocks(mtx,name,fptr)
void CalcRTNR(integer c)
void CalcTCR(integer c)
void CalcCSI(integer c)
void CalcLECTCR(short integer TCR[])
void PrintPair(eoa,eob,c,regress1,regress2,
Bd,Ba,Bi,Be,T,B)
```

from fo_lecam.c

```
void Loadoverloads (* char)
void print_info (void)
void MakeRmultMatrix (void)
```

from matching.c

```
double g(double)
double h(double)
```

from calcb.c

```
void CalcB(integer, integer, integer, * long double, *
integer, * long double, * long double, * long
double, * long double)
```

from calcbigb.c

```
void CalcBigB(* [ 5] [ 3] double, integer, integer)
void LoadTMat(integer)
integer GetEOandT(* integer, * integer, * integer,
integer, * long double, * integer, integer, integer,
integer)
integer TrafBin(long double)
```

from calct.c

```
void CalcT(integer, integer, integer, * long double, *
integer)
```

from selecteoregress.c

```
void SelectEOregress(* integer, * integer, integer,
integer, integer)
```

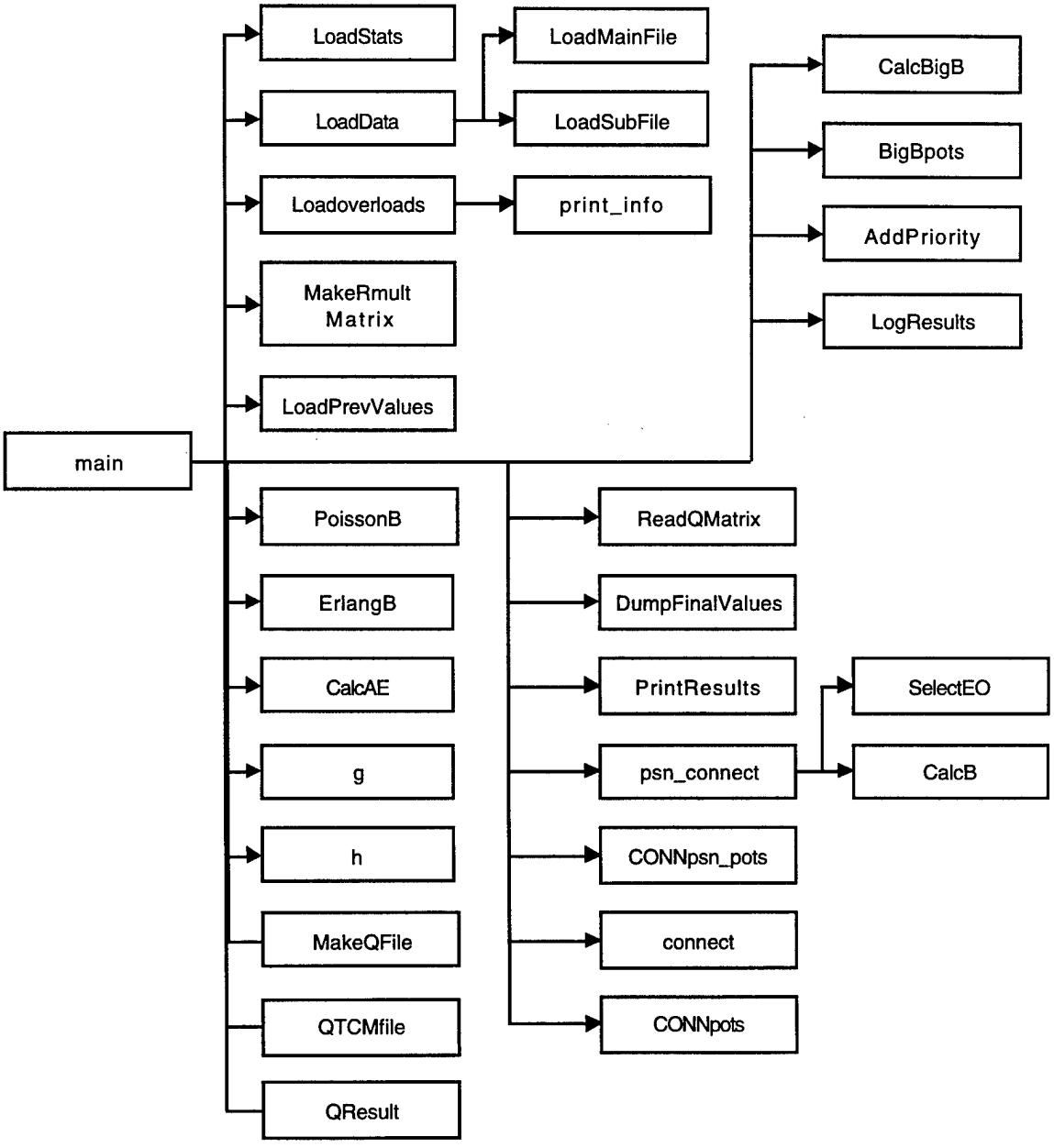
from priority.c

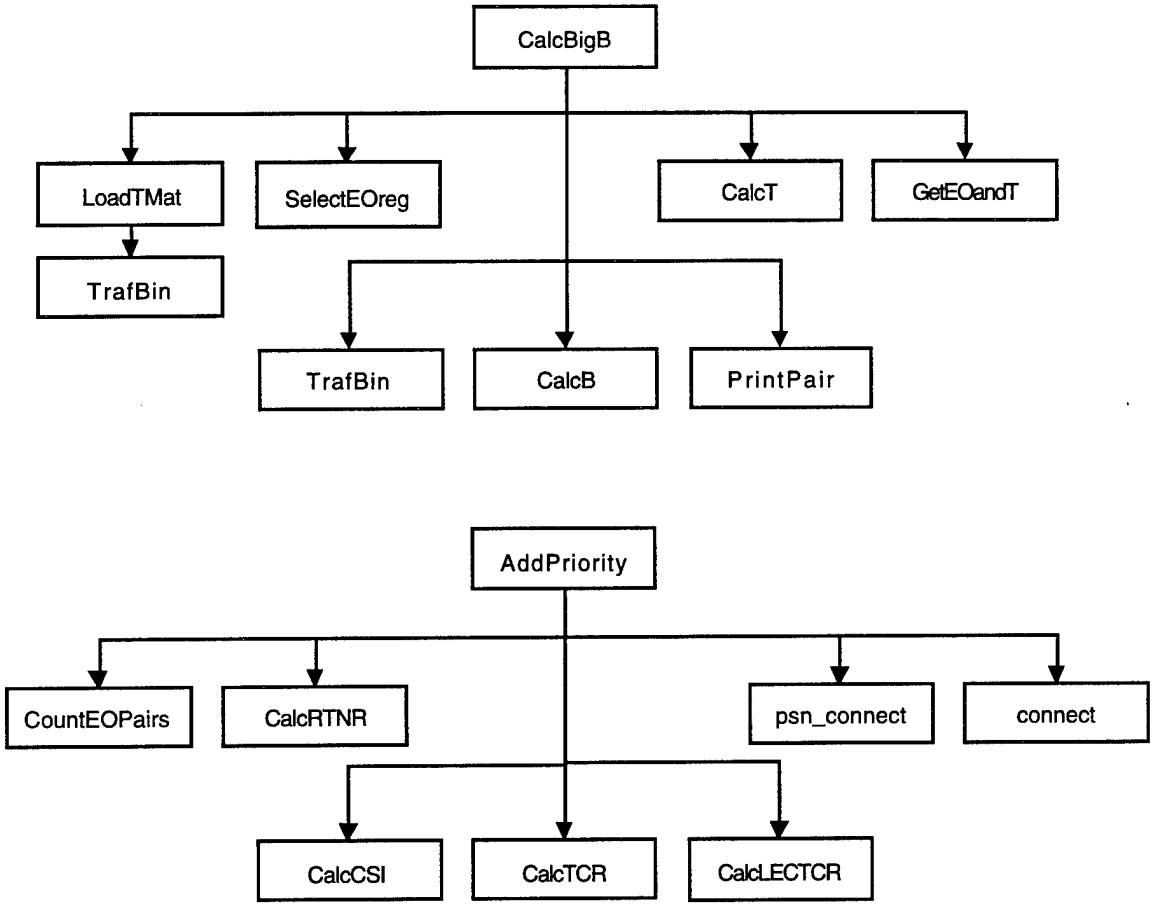
```
void AddPriority(void)
void CountEOPairs(* long)
```

from connect.c

```
double psn_connect(void)
void SelectEO(* integer, * integer)
double connect(integer)
```

Function Tree





Algorithmic Description

This is the functional heart of TAMI. `LECAM` calculates and massages traffic based on requested parameters. It invokes QTCM to calculate IEC and LEC trunk blockages. It iterates on traffic and blockage levels until it reaches steady state. It gathers and processes statistics on traffic and blockages.

`LECAM` first processes command line arguments. It then loads data from IEC and LEC networks, passing the name of the `input_file` to `LoadData()`. If a regional analysis is being performed, overloads are then loaded and the multiplier matrix calculated using `Loadoverloads()` and `MakeRmultMatrix()`.

The next section of `LECAM` performs a number of calculations to initialize traffic and blockage values. If `InitPrev` is set, `LECAM` seeks previous final values, and uses those as an initialization set. These values are retrieved in `LoadPrevValues()`. The variable `fptr`, which is passed into `LoadPrevValues()`, is a FILE pointer, initialized to the hardcoded "TAMI.finalvalues". The code sections until the label `AA:` specify adjustments to base traffic levels based on global overloads, damaged locations, regional overloads and high usage bypasses.

The code segments which follow, up to the comment stating "Calculate sum of AC1 and F0dmg by region," comprise the central iterations of TAMI. These code pieces are organized in an inner and outer loop. The inner loop begins at the label `AA1:` and ends at the statement "`fprintf(stderr, " > Running QTCM for each IEC.`" This loop tests the LEC-IEC interface, attempting to converge on egress traffic. This is logically equivalent to convergence on blockages. At each iteration the inner loop adjusts traffic based on DOC and regional factors. It also uses `CalcAE()`,

a local routine, to determine LEC-IEC trunk traffic flows where trunk glare is a factor. In calculating blockages, this inner loop makes great use of the routines `PoissonB()` and `ErlangB()` from `traffic.c`. The inner loop also has several startup facilities controlled by the `Firstloop` variable. At each iteration, the inner loop seeks convergence on `e2` and `e2new` - the previous and current egress traffic measures. It also seeks to determine if the egress traffic variables are oscillating. It abandons the loop and prints a warning message if it detects such oscillations. In the event that neither oscillation nor convergence are detected, the program branches (using goto statements) back to `AA1:`.

The outer loop is structured much like the inner loop. It tests for convergence or oscillation on `SWBlock` - the blockage at switches. In each iteration it initializes variables, executes the inner loop, invokes QTCM to determine IEC switch blockages, tests for convergence or oscillation, and branches to the label `AA:` (using goto statements) if further iterations are necessary. QTCM is invoked through several steps: creating a TAMI.qtcm_inp from hardcoded data; creating a carrier-specific qtcm file using `MakeQFile()`; invoking a carrier-specific version of QTCM (`qmod_att, qmod_mci, qmod_spr`); and finally retrieving the results generated by QTCM in the routines `QResult()` and `ReadQMatrix()`.

Following the outer loop, lecam produces a number of reports and data files based on the calculated traffic and blockage levels. A seed for the next iteration is generated in the routine `DumpFinalValues()`, which directs this information to a file. The routine `PrintResults()` generates screen output. The next section of code calculates connectivity statistics, calling the routine `connect()`. The final code segment samples EO pairs to determine end-to-end blockages. The sampling and calculations are performed in `CalcBigB()`. `AddPriority` calculates the same blockage numbers including priority routing mechanisms, as determined by switches set by the command line arguments. `LogResults()` prints these end-to-end and connectivity statistics to a running log file. `LECAM` then exits gracefully.

Inputs

FILE	`*fptr`	input file

Outputs

returns no formal returns

Globals:
`ctype` `offdtraffic`

Purpose This routine writes selected results to the indicated file.

Called By `main()`

Calls To library functions
`PrintBlocks()`

Local Variables

real	`trafsum`	placeholder for summary statistic
	`total`	placeholder for summary statistic
	`engineeredtraf`	placeholder for summary statistic
	`comptraf`	placeholder for summary statistic
	`x, y`	placeholder for summary statistic
integer	`i, j, c, k, L, P, S`	loop variable
	`flag`	

Global Variables

integer
`EARLY`
`RTOG`
`NUMREGIONS`
`STOREDOC`
`NSEPMAT`
`CSI`
`LECTCR`
`ol_oscillation`
`e2_oscillation`
`b_oscillation`
`OE_SIZE`
`AT_SIZE`
`IEC_SIZE`
`POP_SIZE`
`POPalive`
`SWITCH_SIZE`
`SWalive`
`HUPOP`
`SD`
`ATPOPSIZE`
`HAP`
`SAP`
`SEA`

kctype	AC1
	AC1prime
	AC11
kc2type	FO
cStype	EG12
	EG1final
character	IECid
real	CONNpsn_pots
	CONNpsn_csi
	CONNpsn_rtnr
Rrctype	BigBpots
	BigBrtnr0
	BigBrtnr1
	BigBcsi
	BigBlec1
	BigBtcr1

Algorithmic Description

`LogResults()` prints the final results of the congestion analyses to the file indicated by `fptr`. It also calculates summary statistics in place.

Inputs

character `infile`

Outputs

returns no formal returns

Purpose This routine loads file names from the keyfile specified in the command line of `tami.c`.

Called By `main()`

Calls To `LoadMainFile(), LoadSubFile()`

Local Variables

integer	`c`	IEC index counter
FILE	`*fptr`	points to the primary input file

Global Variables

char	`IECfiles`	names of IEC inputfiles

Algorithmic Description This routine is a programming interface piece that points to the two routines that load all the basic data for `LECAM`; `LoadMainFile()` and `LoadSubFile()`.

Inputs

FILE * fptr

Outputs

returns no formal returns

Globals:

int IEC_SIZE
AT_SIZE
EO_SIZE
EOAT
SEA
EOalive
ATalive
EOlata

char IECid
IECfiles

Purpose This routine loads global definition data. It also provides `LoadData()` with the names of the IEC specific data files.

Called By `LoadData()`

Calls To library functions

Local Variables

character	line[100]	used to parse a line of input
	temp[10]	used to parse a line of input
integer	i, j, k, c, L, count, n	a variety of counters
double	x	place holder for average taking

Global Variables

char MainTemplate

Algorithmic Description `LoadMainFile()` reads data in the file indicated by `fptr`. The data is partitioned into nine sections, each labeled. The global `MainTemplate` is an array of strings, initialized with the section headings for the main data file, in the proper order. The primary loop in this section is over the sections. The switch statement is used as a jump table, with specific code for each section, and the index dictated by the `MainTemplate` entry corresponding to the current section header. The parsing of individual sections is straightforward, and well commented in the code.

Inputs

integer	c	carrier index
character	filename[ ]	name of the data file

Outputs

returns no formal returns

Globals:

kctype	AC1
	AC1Prime
int	POPSwitch
cStype	ACS
int	SAP
cLPtype	ELAP
kctype	FOdmg
kcLtpe	FO
int	SD
	HAP
	ATPOP
	HUPOP
	Switch_SIZE
	POP_SIZE
	AT_SIIZE
	EO_SIZE
	ATPOP_SIZE
	POPalive
	SWalive
	PhysDiv
cSStype	SS
	SSTNG
char	QTCMfile

Purpose This routine loads carrier-specific information from the named file.

Called By LoadData()

Calls To library functions

Local Variables

FILE	*fptr	
character	line[100]	
	temp[10]	
integer	i, j, k, c, L, count, n	a variety of counters
double	x, y	helpers for averaging

Global Variables

int `AT_SIZE`
`EO_SIZE`

Algorithmic Description

`LoadSubFile()` reads data in the file indicated by `filename` for the carrier indicated by `c`. The data is partitioned into 22 sections, each labeled. The global `SubTemplate` is an array of strings, initialized with the section headings for the carrier data file, in the proper order. The primary loop in this section is over the sections. The switch statement is used as a jump table, with specific code for each section, and the index dictated by the `SubTemnplate` entry corresponding to the current section header. The parsing of individual sections is straightforward, and well commented in the code.

Inputs

FILE `* fptr`

Outputs

returns no formal returns

Globals:
Ltype `OLAT`
ktype `OLEO`
cLPtype `e2`
cSStype `SWBlock`

Purpose This routine loads ending values from a previous iteration as starting values for the current iteration.

Called By `main()`

Calls To library functions

Local Variables

FILE	`* fptr`	
character	`line[ 50]`	
integer	`c, L, P, k, i, PM, S`	a variety of counters
double	`x, y`	helpers for averaging

Global Variables

int `EO_SIZE`
`AT_SIZE`
`IECSIZE`
`ATPOPSIZE`

Algorithmic Description `LoadPrevValues()` loads data from the file pointed to by `fptr` for the LECs and IECs. This file is formatted in three major sections, relating to LEC ATs, LEC EOs and IEC data. Within the IEC data section there are subsections for each IEC, containing data on AT-POP trunk blockages, and switch blockages.

Inputs

float	b	trunk blockage
	AC	access traffic
	EG	egress traffic

Outputs

returns no formal returns

float	*a	final access blockage
	*e	final egress blockage

Purpose This routine caculates the proportion of access and egress traffic blocked, as determined by glare resolution

Called By `main()`

Calls To `none`

Local Variables

float	x	placeholder

Global Variables none

Algorithmic Description The NCS Traffic Congestion Analysis Report (Nov. 92) contains detailed derivations of the glare equations.

Inputs

int	c	carrier index
char	qtcm_file	template file name

Outputs

returns no formal returns

Purpose This routine creates a QTCM input file based on the template file qtcm_file and the switch-to-switch traffic matric AC14.

Called By main()

Calls To library functions

Local Variables

int	i, j, count	loop variables
char	line[100]	working storage
FILE	*inptr, *outptr	input and output files

Global Variables

cSStype	AC14	switch-to-switch traffic

Algorithmic Description MakeQFile() generates a QTCM input file, title TAMI.lecam.qtcm.tmpfile. This file is created based on template file, referenced by qtcm_file. The qtcm_file template's sections 7 is replaced by data from the current iteration, for the carrier indicated by the parameter c. This data is taken from the global variable AC14. The data is output in a transposed matrix to match QTCM's input format.

Inputs

char	`fname`	name of the ppmeanfile to be read.

Outputs

returns	float	aggregate blockage from fname

Purpose This routine reads the QTCM `ppmeanfile` for the results of the final QTCM run.

Called By `main()`

Calls To library files

Local Variables

FILE	`*fptr`	input file
char	`temp[ 8] , line[ 80]`	working storage
float	`b`	result from the file

Global Variables none

Algorithmic Description Opens and closes the indicated `ppmeanfile`. This routine reads the QTCM final result from the first line.

Inputs

csstype	SWBlock	switch-to-switch blockages
int	IEC	IEC index
char	fname	name of the desired ppmeanfile

Outputs

returns no formal returns

Globals:
cSStype SWBlock

Purpose ReadQMatrix() reads the switch-to-switch blockages produced by QTCM from the indicated ppmeanfile.

Called By main()

Calls To library functions

Local Variables

FILE	*fptr	input file
char	temp[8] , line[100]	working storage
int	i, j, count	loop variables

Global Variables

int	SWITCH_SIZE	the number of switches in any give IEC

Algorithmic Description Opens and closes the indicated ppmeanfile. The first line is skipped, then this routine reads the QTCM result from the remaining lines.

Inputs none

FILE	`*fptr`	output file

Outputs

returns no formal returns

Purpose `DumpFinalResults(1)` dumps results from the current iteration for use in later iterations.

Called By `main()`

Calls To library functions

Local Variables

FILE	`*fptr`	input file
int	`c, L, P, k, i, PM, S`	loop variables

Global Variables

int	`ATPOPSIZE`	the number of ATPOP trunks in any give IEC
	`SWITCH_SIZE`	the number of switches in any given IEC
Ltype	`OLAT`	
ktype	`OLEO`	
cLPtype	`e2`	
cSStype	`SWBlock`	

Algorithmic Description `DumpFinalResults()` is a simple data dump. It opens the file "`TAMI.finalvalues`" and outputs `OLAT, OLEO, SWBlock` and `e2`. This data dump is of a format to be reread by `LoadFinalValues()`.

Inputs none

Outputs

returns no formal returns

Purpose This routine prints important variables to the screen.

Called By `main()`

Calls To library functions

Local Variables none

Global Variables

int	`DOCLEVEL`
	`DOC`
	`EARLY`
	`bcount`
	`totale2count`
	`totalOLcount`
float	`Tmult`
kctype	`e0`
	`a0`
	`a3`
ktype	`e1`
	`a1`
cLPtype	`e2`
	`a2`
int	`EOalive`
	`ATalive`
	`AT_SIZE`
	`IEC_SIZE`
	`ATPOPSIZE`
Ltype	`OLAT`
ktype	`OLEO`

Algorithmic Description This routine prints important variables to the screen.

Inputs

FILE	`*fptr`	input file

Outputs

returns no formal returns

Globals:

ctype `offdtraffic`

Purpose This routine writes selected results to the indicated file.

Called By `main()`

Calls To library functions
`PrintBlocks()`

Local Variables

float	`trafsum`	input file
	`total`	
	`engtraf`	
	`comptraf`	
	`x, y`	
int	`i, j, c, k, L, P, S`	loop variable
	`flag`	

Global Variables

int
`EARLY`
`RTOG`
`NUMREGIONS`
`STOREDOC`
`ol_oscillation`
`e2_oscillation`
`b_oscillation`
`EO_SIZE`
`AT_SIZE`
`IEC_SIZE`
`POP_SIZE`
`POPalive`
`SWITCH_SIZE`
`SWalive`
`HUPOP`
`SD`
`ATPOPSIZE`
`HAP`
`SAP`
`SEA`

kctype
`AC1`
`AC1prime`
`EG12`

kc2type	FO
cStype	AC11
	EG1final
char	IECid
float	CONNpsn_pots
	CONNpsn_csi
	CONNpsn_rtnr
Rrctype	BigBpots
	BigBrtnr0
	BigBrtnr1
	BigBcsi
	BigBlec1
	BigBtcr1
int	NSEPMAT
	CSI
	LECTCR

Algorithmic Description

`LogResults()` prints the final results of the congestion analyses to the file indicated by `fptr`. It also calculates summary statistics in place.

Inputs

RRctype	`mtx`	blockage matrix
char	`name`	name of summary block
FILE	`*fptr`	output file

Outputs

returns no formal returns

Purpose This routine formats and prints blocked traffic information to the indicated file.

Called By `LogResults()`

Calls To library functions

Local Variables

int	`reg1, reg2`
float	`x`

Global Variables

ctype	`offdtraf`
int	`NUMREGIONS`
	`IEC_SIZE`
char	`IECid`

Algorithmic Description `PrintBlocks()` receives blocking information passed in from `LogResults()` and combines this blockage information with global traffic information. This information is then output to the indicated file.

Inputs

integer	*A	pointer to originating end office
	*B	pointer to terminating end office

Outputs

returns no formal returns

Purpose to select two random EOs form different LATAs

Called By
```
connect()
psn_connect()
```

Calls To

drandom()	returns a random value

Local Variables

integer	lata	holds an EO lata value
double	number	holds a random number returned from drandom()

Global Variables

RSTREAM	a random number stream
EO_SIZE	size of the EO
EOlata	EO to LATA connection

Algorithmic Description SelectEO returns two EOs from different LATAs

Inputs

integer	c	indicates IEC

Outputs

returns	float	a psn connectivity value

Purpose to determine connectivity through one IEC

Called By main()

Calls To

CalcB	calculates blockages
SelectEO	selects two EOs from different LATAs

Local Variables

integer	c	loop count variable
	target	passed in vlaue of TargetKey[ ]
	sum	running total of number of switch pairs
	count	loop count variable
	EOA	originating end office
	EOB	terminating end office
	flag	pointer flag
	idx	index value
	test	boolean flag
double	B	overall blocakge
	bd	doc blockage
	ba	access blocakge
	bi	iec blockage
	be	egress blockage
float	pp	percentage of connectivity

Global Variables

integer	TargetKey[11]	target number of EO pairs
	HUPOP	high usage trunk to POP connectivity value
	ATPOP	AT to POP connectivity value
	EO_SW	EO to switch connectivity value

Algorithmic Description This routine calculates a single IEC connectivity number. An EO pair is considered connected if it has connectivity through the specified IEC (c). This routine differs from psn_connect() in that it looks for connections through one IEC only.

Inputs none

Outputs

returns	float	a psn connectivity value

Purpose to determine connectivity through all IECs

Called By `main()`

Calls To `CalcB`
`SelectEO`

Local Variables

integer	`c`	loop count variable
	`target`	passed in vlaue of `TargetKey[ ]`
	`sum`	running total of number of switch pairs
	`count`	loop count variable
	`EOA`	originating end office
	`EOB`	terminating end office
	`flag`	pointer flag
	`idx`	index value
	`test`	boolean flag
double	`B`	overall blocakge
	`bd`	doc blockage
	`ba`	access blocakge
	`bi`	iec blockage
	`be`	egress blockage
float	`pp`	percentage of connectivity

Global Variables

integer	`TargetKey[ 9]`	target number of EO pairs
	`HUPOP`	high usage trunk to POP connectivity value
	`ATPOP`	AT to POP conncectivity value
	`IEC_SIZE`	size of the IEC

Algorithmic

Description This routine calculates a PSN connectivity number. An EO pair is considered connected if it has connectivity through at least one IEC. This routine differs from `connect()` in that it looks for connections through all IECs.

Inputs

integer	c	IEC id number

Outputs

returns	no formal returns	
Globals:		
cSStype	SWBlock	returns the updated switch-switch blockages

Purpose For IEC c (should be AT&T), this routine converts the corresponding switch-to-switch blockage matrix (SWBLOCK) from direct-connect-only blockage to an alternate-routing blockage.

Called By AddPriority()

Calls To library functions

Local Variables

real	x1, x2, x3, x4	place holders
integer	i, j, k	loop variable
	count1, count2	
	count3, count4	

Global Variables

integer	SWalive
cSStype	SWBlocknew
integer	SWITCH_SIZE

Algorithmic Description CalcRTNR adjusts the switch-switch blockage matrix to account for RTNR. For each switch, and for each of its connections to other switches, CalcRTNR considers all available alternative routes. The appropriate adjustment between switch I and J via switch K is:

```
(1 - (1-SWBlock[ c][ I][ K] )*(1-SWBlock[ c][ K][ J] ) ).
```

There are appropriate sanity checks to rule out vias through dead switches, or dead origin/destination switches.

Inputs

integer	c	IEC id number

Outputs

returns no formal returns

Globals:
cSStype SWBlock

Purpose This routine adds IEC trunk congestion relief (TCR) to the IEC c switch-switch blockage matrix (SWBlock).

Called By AddPriority()

Calls To library functions

Local Variables

real	x, sum	place holder variables
integer	i, j	loop variable
	count, count2	

Global Variables

integer SWITCH_SIZE

Algorithmic Description Any SWBlock entry which is not 1.0 (representing 100% blockage) is changed to 0.0 (representing 0% blockage).

Inputs

integer	c	IEC id number

Outputs

returns no formal returns

Globals:
cSStype SWBlock

Purpose This routine alters the AT&T SWBlock matrix to reflect CSI trunks for NSEP traffic.

Called By AddPriority()

Calls To library functions

Local Variables

character	line[50]	working storage
integer	a, z, surv count1, count2	loop variable
FILE	*fptr	input file pointer

Global Variables

character	CSIfile
integer	SWITCH_SIZE

Algorithmic Description

The CSI data is loaded from the file named in CSIfile. Each record is parsed for an originating switch index (zero-based), a destination switch index, and a damage vector (0/1). If a link survives damage (1) then the SWBlock matrix between the two switches is reduced to zero (no blockage). All other blockages remain unchanged.

Inputs

short integer TCR

Outputs

returns no formal returns

Globals:

kctype	a3
	a0
	e0
ktype	a1
	e1
cLPtype	a2
	e2
ktype	OLEO
Ltype	OLAT

Purpose This routine zeros LEC blockage for all EOs in selected LATAs.

Called By AddPriority()

Calls To library functions

Local Variables

integer	k, c, PM	loop variable

Global Variables

integer
EOlata
EOAT
HUPOP
ATPOP
ATPOPSIZE
IEC_SIZE
EO_SIZE

Algorithmic Description CalcLECTCR zeros blockages associated with LEC facilities in the LATAs indicated by the TCR parameter. Overloads at affected EOs and ATs are also zeroed.

Inputs

integer	`eoa, eob, c, reg1, reg2`
long double	`Bd, Ba, Bi, Be, T, B`

Outputs

returns no formal returns

Purpose This routine writes selected results to the standard input, largely for debugging.

Called By `CalcBigB()`

Calls To library functions

Local Variables

integer	`s1, s2`	switch indexes

Global Variables

character	`IECid`
kctype	`AC1prime`
	`AC1`
cStype	`TRAF_DELT`
	`TRAF_KEEP`
cSStype	`SSTNG`
	`SSreg`
	`AC14adj`
	`AC14`

Algorithmic Description This routine does no computation; it simply prints certain statistics about an EO-EO pair.

Inputs

character	`*infile`	input file

Outputs

returns: no formal returns

Globals:

real	`Rmult`	regional traffic multiplier
	`Lmult`	LATA traffic multiplier
integer	`LReg`	LATA to region mapping
	`EOReg`	EO to region mapping

Purpose This routine loads regional information.

Called By `main()`

Calls To library functions
`print_info(void)`

Local Variables

double	`multiplier`	
character	`buffer1[80], buffer2[80]`	working memory
integer	`i,lata, region`	loop variable
FILE	`*inputfile`	input file

Global Variables

integer	`NUMREGIONS`
	`NUMEO`
	`EOlata`
	`EO_SIZE`

Algorithmic Description `Loadoverloads()` first opens the indicated file. For each region, it then retrieves the region designation and multiplier. It then reads one line for each LATA in the region. The overload and region designation for the lata is then set. `Loadoverloads()` then closes the file, and sets the region for each EO.

Inputs none

Outputs

returns no formal returns

Purpose This routine writes progress indicators to stdout.

Called By `Loadoverloads()`

Calls To library functions

Local Variables

integer	`i`	loop variable

Global Variables

real	`Rmult`	regional traffic multiplier
	`Lmult`	LATA traffic multiplier
integer	`LReg`	LATA to region mapping
	`EOReg`	EO to region mapping
	`NUMREGIONS`	

Algorithmic Description This routine loops through all available lata and regions, printing the region/multiplier pairs.

Inputs none

Outputs

returns no formal returns

Globals:
real `RmultMatrix`

Purpose This routine generates a matrix of region to region traffic multipliers..

Called By `main()`

Calls To library functions

Local Variables

integer `i, j` loop variable

Global Variables

real `Rmult`

Algorithmic Description This routine builds a symmetric region to region traffic multiplier matrix. The region to region traffic multiplier is taken as the max of the two regional multipliers.

Inputs

real	OL	the traffic overload at the switch

Outputs

real	temp	outgoing (access) loss at the switch

Purpose This returns the access matching loss at a switch.

Called By main()

Calls To none

Local Variables

real	temp

Global Variables

integer	MATCHOFF HIGHMATCH SETMATCH
real	SETMATCHLEVEL

Algorithmic Description This routine determines an access matching loss based on a step-wise linear function. There are several variants available through command-line settings (-D -E -F).

Inputs

real	OL	switch overload

Outputs

real	temp	incoming (egress) matching loss

Purpose This returns the egress matching loss at a switch.

Called By main()

Calls To none

Local Variables

real	temp

Global Variables

integer	MATCHOFF HIGHMATCH SETMATCH
real	SETMATCHLEVEL

Algorithmic Description This routine determines an egress matching loss based on a step-wise linear function. There are several variants available through command-line settings (-D -E -F).

Inputs

integer	`eoa, eob`	
	`c`	IEC id number
	`*flag`	returns error indicator flag
double	`*b`	returns overall blockage
	`*bd`	returns blockage due to DOC
	`*ba`	returns access blockage
	`*bi`	returns IEC blockage
	`*be`	returns egress blockage

Outputs

returns no formal returns

Globals:

Purpose calculates EO to EO blockage.

Called By
```
psn_connect()
connect()
CalcBigB()
CountEOPairs()
```

Calls To library functions
```
g()
h()
```

Local Variables

real	`mult`
double	`x, y, a, e, a3`
integer	`L, P, M, S1, S2, test`
	`Atype, Btype`
	`EOATa, EOATb`
	`EOSWa, EOSWb`
	`HUPOPa, HUPOPb`
	`ATPOPa, ATPOPb`

Global Variables

integer
```
EOAT
EOSW
HUPOP
ATPOP
EOalive
EARLY
POPalive
SWalive
ATalive
PhysDiv
HAP
```

	ATPOPSIZE	
	DOC	
	RTOG	
	EOReg	
kctype	a0, e0	the following are used to determine DOC levels
ktype	OLEO	
Ltype	OLAT	
cLPtype	a2, e2	
ktype	a1, e1	
cSStype	SWBlock	
	SSTNG	
	SS	
real	RmultMatrix	
	Tmult	
cStype	EG1	
	ACSdmg_nomult	
	ACSlec_live	
kctype	EG2	
	AC1prime	
	F0dmg	
kc2type	F0	

Algorithmic Description

`CalcB` calculates EO to EO total blockage. It first determines the LEC to IEC topology, adjusting for damage. If the EOs in question are not connected to each other, 1.0 is returned for all blockages and the `flag` variable is set appropriately. Globally available blockage figures are then combined appropriately to yield access, egress and IEC blockages. These are then combined to determine overall blockage.

Inputs

RRctype	BigB	returns blockages
integer	c	IEC id number
integer	TRAFMAT	flag indicating whether or not a traffic matrix has been provided

Outputs

returns no formal returns

Purpose This routine uses statistical sampling to estimate the overall EO-EO blockage, based on minimal EO pairs.

Called By
```
main()
AddPriority()
```

Calls To
```
library functions
LoadTMat()
SelectEOreg()
CalcT()
GetEOandT()
TrafBin()
CalcB()
PrintPair()
```

Local Variables

integer	target[MAX_BIN]	target sample count
	count[MAX_BIN]	current sample count
	reg1, reg2, eoa, eob	
	flag, k, i	
integer	bin_count	
	bin	
	tot_pairs	
	bin_flag[MAX_BIN]	is the bin filled
	abort, n, test	
	OutOfPairs	
double	num[MAX_BIN]	
	denom[MAX_BIN]	
	B, T	
	sumb2t[MAX_BIN]	
	x, y	
	mean[MAX_BIN, sdev[MAX_BIN]	
	error[MAX_BIN] , rp[MAX_BIN]	
	Ba, Bi, Be, Bd	
	numBa[MAX_BIN]	
	numBe[MAX_BIN]	
	numBi[MAX_BIN]	
	numBd[MAX_BIN]	
	xD	

Global Variables

integer
- BUNDLE_INC
- EOReg
- EOSW
- EO_SIZE
- BUNDLE_SIZE
- EOlata
- NUMREGIONS
- HUPOP
- ATPOP

from calcbigb.c

integer
- reg1_eolist
- region
- num_samples
- INMEM
- reg2_eolist
- reg1_tot:
- Tijptr
- reg2_tot
- testbin

double
- testT

Algorithmic Description

`CalcBigB` uses statistical sampling to estimate traffic-weighted EO-EO blockage. If a traffic matrix is given, `CalcBigB` calls `GetEOandT()` to retrieve EOs and traffic values. If no traffic matrix is given, it randomly determines EOs to sample, and uses `CalcT()` to determine the traffic between the selected EOs. This sampling is performed for each region-region pair. The EO pairs are sorted into bins based on traffic values, in order to decrease the probability of rounding inaccuracy. Each bin has an initial target number. Sampling continues until the bin target has been met, or the available EOs are exhausted. At this point, a number of statistics are determined for the sample, including mean, standard deviation and error. If the error is not within acceptable bounds (currently hardcoded at 3 percent), further sampling is performed. The final blockage values are returned in the `BigB` argument.

Inputs none

Outputs

returns no formal returns

Globals:
from calcbigb.c
struct pair `eopair`

Purpose This routine loads a traffic matrix from the file indicated by the variable `nij_file`.

Called By `CalcBigB()`

Calls To library functions
`TrafBin()`

Local Variables

integer `i, count`
character `line[ 80]`
FILE `*nijptr`
integer `histogram[ MAX_BIN]`
double `Tij, sumT[ MAX_BIN]`

Global Variables

character `nij_file`
integer `IECid`
`IEC_SIZE`
from calcbigb.c
integer `region`
`bin_counter`
`num_samples`

Algorithmic Description `LoadTMat()` opens the file indicated by the global variable `nij_file`, and reads the values into the global array `eopair`.

Inputs

integer	`*eoa, *eob`	returns sampled end offices
	`*flag`	returns error if out of samples
double	`*T`	returns traffic
integer	`c`	indicates carrier of interest
	`reg1, reg2`	restricts EO selection

Outputs

integer unused

Purpose returns an EO pair, and the traffic between them.

Called By `CalcBigB()`

Calls To none

Local Variables

integer `found`

Global Variables

integer `EOReg: 183`
`EOSW: 3`

from calcbigb.c
struct pair `eopair`
integer `Tijptr: 533`
`num_samples: 505`

Algorithmic Description `GetEOandT` returns the next sample pair and traffic value from the samples read in `LoadTMat()`. It returns an error if there are no sample pairs left.

Inputs

double `traffic`

Outputs

integer `bin` the bin appropriate to that traffic level

Purpose This routine assigns a bin based on traffic level

Called By `LoadTMat()`
`CalcBigB()`

Calls To library functions

Local Variables

integer `bin`

Global Variables none

Algorithmic Description `TrafBin` encapsulates the knowledge of bin assignment based on traffic value.

Inputs

integer	eoa, eob	
	c	
double	*Tij	returns traffic value
integer	*flag	flag is set if the EOs are not sufficiently connected based on the scenario under consideration

Outputs

returns no formal returns

Purpose This routine determines the POTS traffic between EOs eoa and eob.

Called By CalcBigB()

Calls To library functions

Local Variables

integer	L, P, M, test	loop variable
	Atype, Btype	topology type
	EOATa, EOATb	
	EOSWa, EOSWb	
	HUPOPa, HUPOPb	
	ATPOPa, ATPOPb	
real	mult	

Global Variables

integer	EOAT
	EOSW
	HUPOP
	ATPOP
	EOlata
	EOalive
	EARLY
	POPalive
	POPSWITCH
	SWalive
	ATalive
	PhysDiv
	RTOG
	EOReg
cSStype	SWBlock
	SS: 154
real	RmultMatrix
	Tmult
cStype	ACSdmg_nomult
kctype	F0dmg
	ACSlec_live
kc2type	F0

Algorithmic Description

`CalcT()` determines the POTS traffic between `eoa` and `eob` which is carried on IEC `c`. It first determines the topology of the given EOs, and their connectivity. If they are not sufficiently connected based on the scenario under consideration (`EARLY` or `LATE`), `Tij` is set to 1.0 and `flag` is set. Otherwise, the overload multiplier is set based on the already determined regional and global overloads. Based on the scenario under consideration the `FO` traffic is modified appropriately, and returned in `Tij`.

Inputs

integer	*EOA, *EOB	returns the selected EOs
integer	c	the carrier to which the EOs must be homed
integer	regA, regB	the regions from which to select EOs

Outputs

returns no formal returns

Purpose This routine randomly selects a pair of EOs from different LATAs, fulfilling regional and carrier homing restrictions.

Called By CalcBigB()

Calls To library functions

Local Variables

double	rnumber		holder for random number
integer	n		holds the lata number of EOA
	stream	-15	the random number stream to be used

Global Variables

integer RSTREAM
EOlata

from calcbigb.c
integer reg1_eolist, reg1_tot
reg2_eolist, reg2_tot

Algorithmic Description SelectEOReg() iteratively picks random EOs from the global reg1_eolist and reg2_eolist until the conditions are met. The EOs are returned through *EOA and *EOB.

Inputs none

Outputs

returns no formal returns

Globals:

real
```
CONNpsn_rtnr
CONNpsn_csi
```
ctype
```
CONNrtnr
CONNcsi
```
RRctype
```
BigBrtnr0
BigBrtnr1
BigBcsi
BigBtcr0
BigBtcr1
BigBlec0
BigBlec1
```

Purpose This routine calculates final blockages after priority treatments.

Called By `main()`

Calls To library functions
```
CountEOPairs()
CalcRTNR()
CalcC$I()
CalcTCR()
CalcLECTCR()
```

Local Variables

integer `c, i`

Global Variables

integer
```
COUNTPAIRS
attnum
RSTREAM
IEC_SIZE
DOC
NSEPMAT
CSI
LECTCR
```
character `IECid`

Algorithmic Description `AddPriority()` calculates cumulative benefits from priority treatments. Because `CalcRTNR()`, `CalcCSI()`, etc. alter the `SWBlock` matrix, all calculations are cumulative. Each block of the function calculates an additional enhancement by calling the appropriate `Calc*` function to change the `SWBlock` matrix, then rerunning `CalcBigB()` - perhaps once for each carrier. If a traffic matrix is available, the blockages are also calculated using weighting from the provided traffic matrix.

Inputs

long integer `bins`

Outputs

returns no formal returns

Globals:
ctype `offdtraffic`

Purpose This routine checks connectivity of all EO pairs through each IEC.

Called By `AddPriority()`

Calls To library functions

Local Variables

double `B`
`w, x, y, z`
integer `EOA, EOB`
`i, c, flag`
long integer `count`

Global Variables

integer `IEC_SIZE`
`EOlata`
`EOalive`
`EO_SIZE`

Algorithmic Description For all possible pairs of end offices, their connectivity is assessed through each of the three IECs. `CountEOPairs()` uses `CalcB()` to determine connectivity between each pair, and prints aggregate statistics to `stdout`.

3.12 keepstats: Record Overall Statistics

Purpose This program accumulates TAMI blockage statistics over a number of iterations by maintaining a statistics file. The input statistics for each iteration are given by an input file. This program will accumulate statisticss for each variable in the input file.

Call Syntax

`keepstats` `[ options]`

mandatory:	*special syntax:*	*function:*
none		
options:	*special syntax:*	*function:*
`-k`	<input filename>	reads in input file name, if not specified, default file name is `TAMI.lecamlog`
`-o`	<outfile>	reads in output file name, if not specified, default file name is `TAMI.tally`
`-O`		indicates overwriting of output file
`-?`		user help--prints call syntax and exits without running

example `keepstats -k keyfile -O`

Input Files

TAMI.lecamlog — this file contains output data from the lecam.c module

format **SEE SECTION 3. 11/lecam.c module**

Output Files

`outfile` — The output file for `keepstats` consists of 8 sections which report various blockage calculations and 1 section which reports connectivity. For each iteration, sums are given for the following blockage values for:

- accumulated sums of blockage
- accumulated sums of blockages squared
- number of damage vectors
- accumulated mean blocking sum
- standard deviation

error rate

section1: POTS — <POTS blocakge for each iec, iec = att, mci, spr>

section2: SAMP — <Regional sampling of POTS for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>

section3: RTNR0 — <NSEP traffic with RTNR and DOC off, with POTS weighting for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>

section4: RTNR1 — <NSEP traffic with RTNR and DOC off, with NSEP traffic weighting for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>

section5: TCR0 — <NSEP traffic with TCR and DOC off, with POTS weighting for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>

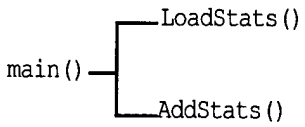
	section6: TCR1	<NSEP traffic with TCR and DOC off, with NSEP traffic weighting for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>
	section7: LEC0	<LEC traffic with DOC off, with POTS weighting for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>
	section8: LEC1	<LEC traffic with DOC off, with NSEP traffic weighting for 2 regions,'0' and '1' for each iec, iec = att, mci, spr>
Includes	`"fileio.c"`	User-defined I/O functions; see Appendix A
	`<stdio.h>`	The standard 'C' I/O functions.
	`<math.h>`	The standard 'C' math functions.
Constants	`FALSE`	`0`
	`MAXPOINTS`	`30`
	`MAXVAR`	`300`
Global Variables		
structure	`StatList` of type `records` *with fields*	
character	`varname[ 30]`	the name of the variable being tallied
real	`sum`	sum of blockage
	`sumsqr`	sum of blockage-squared
	`mean`	mean blockage
	`sdev`	standard deviation of blockage
	`error`	sampling error
integer	`iter`	number of iterations tallied
record	`StatList[ MAXVAR]`	
real	`T[ MAXPOINTS+1] =`	`{ 0.000,12.706,4.303,3.182,2.776,2.571, 2.447,2.365,2.306,2.262,2.228,2.201, 2.179,2.160,2.145,2.131,2.120, 2.110, 2.101,2.093,2.086,2.080,2.074,2.069,2.064, 2.060,2.056,2.052, 2.048,2.045,2.042}`
Local Variables	Variables local to `main()`	
extern	character `*optarg`	a string containing a single command line argument.
integer	`optind`	the number of single command line arguments to be processed, supplied externally by the operating system
	`err=FALSE`	sets command line error indicator off
	`itog=FALSE`	sets command line input file error indicator off
	`otog=FALSE`	sets command line output file error indicator off
	`OVERWRITE=FALSE`	sets command lineoutput file overwrite indicator off
	`n,i`	loop count variables

	numvars	used to count the number of input variables
	maxidx	used to index the input variables
character	ch	a command line option character
	filelist[80]	holds the name of the input key file
	**argv[ ]	an array containing all of the command line arguments
	infile[50]	holds the name of the input variable
	outfile[50]	holds the name of the output variable
	line[100]	holds a line of input
	name[30]	holds an input variable
	namestart[100]	holds the starting position for an input variable
	bigb[30]	holds an input variable
FILE	*inptr	points to the input file
	*outptr	points to the output file
real	m,s,e	not used
	maxerr	maximum allowable sampling error

Component Functions

LoadStats()	reads existing output file and loads value into the record structure.
AddStats()	sums output statistics.

Function Tree



Algorithmic Description

This module is used to print to an outfile, file various TAMI blockage statistics. This program accumulates TAMI blockage statistics over a number of iterations by maintaining a stats file called TAMI.tally (unless a different stats file is given by the command line). The input stats for each iteration are given by a file named TAMI.lecamlog (or other as specified). This program will accumulate stats for each variable in the input file. The variable names must begin with 'BIGB' and be followed immediately by an equal sign and the blockage for that variable, e.g.:

BIGBPOTS=0.12345 or
BIGBatt=0.987655

If an entry in the output file does not already exist for a given variable, then one will be created.

The output file contains the following: var name, sum of blockages, sum of blockages squared, the number of iterations accumulated, the mean blockage, standard deviation, and error. The error for BIGBPOTS will be written to the screen for output back to the shell script.

Std.Dev. Formula:

$$S = \sqrt{\frac{TBS - (M^2 * N)}{N - 1}}$$

where TBS is the total blockage squared, N is the sample size, and M is the sample mean.

Error Formula:

$$E = \frac{T(N-1, C) * S}{\sqrt{N}}$$

where S is the sample standard deviation, N is the sample size, and T is the value of the T-distribution for N-1 and confidence level C. (The data for the T-dist is loaded for N-1 up to 30 and C=95%.

Inputs

character	* fptr	pointer to an input file

Outputs

structure	StatList of type records *with fields*	
character	varname[30]	the name of the variable being tallied
real	sum	sum of blockage
	sumsqr	sum of blockage-squared
	mean	mean blockage
	sdev	standard deviation of blockage
	error	sampling error
integer	iter	number of iterations tallied

returns this module returns the value of i, the number of variables in the Tally file

Purpose For each variable in the input file, this function loads the variable value into the StatList structure.

Called By main()

Calls To none

Local Variables

integer	i	loop count variable

Global Variables none

Global Constants

	MAXVAR	maximum number of variables expected from input file

Algorithmic Description For each variable in the input file, this function loads the variable value into the StatList structure. It also counts the variables to ensure that they don't exceed MAXVAR.

Inputs

integer	`*numvars`	pointer to the number of variables
character	`name[ ]`	temporarily holds a variable name
real	`B`	blocking probablity

Outputs

structure	`StatList` of type `records` *with fields*	
character	`varname[ 30]`	the name of the variable being tallied
real	`sum`	sum of blockage
	`sumsqr`	sum of blockage-squared
	`mean`	mean blockage
	`sdev`	standard deviation of blockage
	`error`	sampling error
integer	`iter`	number of iterations tallied

returns integer

Purpose For each variable in the `StatList` structure, this function adds the next value of that variable to the existing value.

Called By `main()`

Calls To none

Local Variables

integer	`i,j`	loop count variables

Global Variables none

Global Constants none

Algorithmic Description For each variable in the `StatList` structure, this function adds the next value of that variable to the existing value, thus accumulating a total output file.

Appendix A: User-Defined Utility Functions

Functions that are repeatedly utilized by more than one module have been placed in this appendix in order to make them readily available. These "utility" functions are divided into two groups:

1) Function calls repeatedly coded into various modules

```
fget()
char_comp()
```

2) Function calls included in include "fileio.c", "traffic.c", and "poisson.c":

```
fileio.c
   parse()
   parse_int()
   getline()
   fopenfile()

traffic.c
   ErlangA()
   ErlangB()
   ErlangN()
   PoissonA()
   PoissonB()
   PoissonN()

poisson.c
   PoissonAmod()
   PoissonNmod()

phdiv.c
   phdiv()
```

Appendix A: User-Defined Utility Functions

Functions that are repeatedly utilized by more than one module have been placed in this appendix in order to make them readily available. These "utility" functions are divided into two groups:

1) Function calls repeatedly coded into various modules

```
fget()
char_comp()
```

2) Function calls included in include "fileio.c", "traffic.c", and "poisson.c":

```
fileio.c
   parse()
   parse_int()
   getline()
   fopenfile()

traffic.c
   ErlangA()
   ErlangB()
   ErlangN()
   PoissonA()
   PoissonB()
   PoissonN()

poisson.c
   PoissonAmod()
   PoissonNmod()

phdiv.c
   phdiv()
```

file *fopenfile(filename, type) **fileio.c**

This utility function is a simple modification of the standard 'C' `fopen` function. It opens the passed in filename and checks for an error in the file. If an error exists the function is exited.

void parse(start, num, buffer, rtn) **fileio.c**

This utility function reads `num` characters from the input character string `buffer` starting at position `start` and directs the output to the character string `rtn`.

int parse_int(start, num, buffer) **fileio.c**

This utility function reads `num` characters from the input character string `buffer` starting at position `start` and returns the integer value of the characters

int getline(fildes, buf) **fileio.c**

This utility function reads from the file `fildes` until the first end-of-line character is reached, and directs the output to the buffer `buf`

double ErlangA(double Bin, int N) **traffic.c**

This utility function calculates the amount of offered traffic given an ErlangB blocking probability `Bin`, and a trunk size `N`.

double ErlangB(double A, int N) **traffic.c**

This utility function uses ErlangB statistics to calculate a blocking probability given trunk size `N`, and the amount of offered traffic `A`.

int ErlangN(double A, double Bin) **traffic.c**

This utility function calculates trunk size given an Erlang B blocking probability `Bin`, and the amount of offered traffic `A`.

double PoissonA(double Bin, int N) **traffic.c**

This utility function calculates offered traffic given a Poisson blocking probability `Bin`, and the number of trunks `N`.

double PoissonB(double A, int N) **traffic.c**

This utility function uses Poisson statistics to calculate a blocking probability given trunk size `N`, and the amount of offered traffic `A`.

double PoissonN(double A, double Bin) **traffic.c**

This utility function calculates the number of trunks, given a Poisson blocking probability `Bin`, and the amount of offered traffic `A`.

double PoissonAmod(double Bin, int Nin) **poisson.c**

This utility function Calculates offered traffic given a Poisson blocking probability `Bin`, and the number of trunks `Nin`. It is a "reality" modification of the PoissonA routine. It is based on the assumption that TGs with more than 250 trunks are engineered for traffic levels proportional to the level for exactly 250 trunks. Traffic in TGs with fewer than 250 trunks is calculated normally with the PoissonA routine. Two traffic constants have been precalculated for this routine. A005 and A01 represent the traffic levels through 250 trunks which produce blockages of 0.005 and 0.01 respectively.

double PoissonNmod(double Ain, double Bin) **poisson.c**

This utility function calculates the number of trunks, give a Poisson blocking probability `Bin`, and the amount of offered traffic `Ain`. It is a "reality" modification of the PoissonN routine. It is based on the assumption that TGs with more than 250 trunks are engineered for traffic levels proportional to the level for exactly 250 trunks. Traffic in TGs with fewer than 250 trunks is calculated normally with the PoissonN routine. Two traffic constants have been precalculated for this routine. A005 and A01 represent the traffic levels through 250 trunks which produce blockages of 0.005 and 0.01 respectively.

int phdiv(eo,at,pop) **phdiv.c**

This utility function performs the physical diversity model. It returns a 1 (TRUE) if the soecified EO-AT-POP trunk group is physically diverse from the parallel EO-AT-POP trunk group. Otherwise it returns a 0 (FALSE)

Appendix B: keyfile

This appendix contains samples of the key files used to execute a typical run of the TAMI program.

File name: keyfiles.h

```
/*************************************************************************/
/*:                                                                      */
/*  Global File Handling Declarations: global files read from keyfile.  */
/*************************************************************************/
    FILE *fileptr,*fptr,*outfile,*outptr;

        /* LEC */
        char eo_file[80],at_file[80];
        /* IEC */
        char iec_eo_file[80],iec_at_file[80];
        char iec_sw_file[80],iec_pop_file[80];
    char qlink_file[80];
        /* MCI & SPR */
    char morph_file[80];
        /* MCI ONLY */
    char swhmg_file[80];
```

File name: dpkey.94

```
#/*******************************************/
#/* 3/94                                    */
#/* This dataprep keyfile points to fy94 data */
#/*******************************************/

LEC
/wrk/task/psn_cong_94/data/tami/eo_live_file
/wrk/task/psn_cong_94/data/tami/at_live_file

ATT
/wrk/task/psn_cong_94/data/tami/switch_att.data
/wrk/task/psn_cong_94/data/tami/pop_att.data
/wrk/task/psn_cong_94/data/tami/eo_att_file
/wrk/task/psn_cong_94/data/tami/hap.att
/wrk/task/psn_cong_94/data/iec/att/qlink.replaced.att

SPR
/wrk/task/psn_cong_94/data/tami/switch_spr.data
/wrk/task/psn_cong_94/data/tami/pop_spr.data
/wrk/task/psn_cong_94/data/tami/eo_spr_file
/wrk/task/psn_cong_94/data/tami/hap.spr
/wrk/task/psn_cong_94/data/iec/spr/qlink.replaced.spr
/wrk/task/psn_cong_94/data/iec/spr/trafdist94.spr

MCI
/wrk/task/psn_cong_94/data/tami/switch_mci.data
/wrk/task/psn_cong_94/data/tami/pop_mci.data
/wrk/task/psn_cong_94/data/tami/eo_mci_file
/wrk/task/psn_cong_94/data/tami/hap.mci
/wrk/task/psn_cong_94/data/iec/mci/qlink.replace.mci
/wrk/task/psn_cong_94/data/iec/mci/trafdist94.mci
/wrk/task/psn_cong_94/data/iec/mci/sw_hmg.mci
```

Appendix C: makefile

This appendix contains a sample of the file used to compile and link the required code for running the TAMI program.

File name: makefile

```
COPTS= -g

merge:  merge.c traffic.o
        cc $(COPTS) merge.c traffic.o -lm -o merge

#*****************************************************************************/
#* The traffic.c code contains all of the Poisson and ErlangB calculation    */
#* routines used by the AT&T, MCI, and Sprint dataprep routines.              */
#*****************************************************************************/
traffic.o: traffic.c
        cc -c $(COPTS) traffic.c

fo.o: fo.c
        cc -c $(COPTS) fo.c

region.o: region.c
        cc -c $(COPTS) region.c

selecteo.o: selecteo.c
        cc -c $(COPTS) selecteo.c

phdiv.o: phdiv.c
        cc -c $(COPTS) phdiv.c

poisson.o: poisson.c
        cc -c $(COPTS) poisson.c

loadlivekey.o: loadlivekey.c
        cc -c $(COPTS) loadlivekey.c
#*****************************************************************************/
#* The attlive program performs calculations on the live AT&T network and    */
#* produces a live intermediate file for damage by attwdmg.                  */
#*****************************************************************************/
attlive:  attlive.o traffic.o phdiv.o poisson.o loadlivekey.o
        cc $(COPTS) attlive.o traffic.o phdiv.o poisson.o loadlivekey.o -lm -o attlive

attlive.o:  attlive.c
        cc -c $(COPTS) attlive.c

#*****************************************************************************/
#* The attwdmg program inputs the attlive network datafile and a damage      */
#* vector for AT&T and LEC assets. It outputs a completed dataprep file for  */
#* input into LECAM.                                                         */
#*****************************************************************************/
attwdmg:  attwdmg.o fo.o region.o traffic.o selecteo.o /user/gretchen/waglib/waglib.o
        cc $(COPTS) attwdmg.o fo.o region.o traffic.o selecteo.o /user/gretchen/waglib/waglib.o -lm -o attwdmg

attwdmg.o:  attwdmg.c
        cc -c $(COPTS) attwdmg.c
```

```
#*****************************************************************************/
#* The mcilive program performs calculations on the live MCI network and     */
#* produces a live intermediate file for damage by mciwdmg.                  */
#*****************************************************************************/
mcilive:  mcilive.o traffic.o loadlivekey.o poisson.o phdiv.o
        cc $(COPTS) mcilive.o traffic.o loadlivekey.o poisson.o phdiv.o -lm -o mcilive

mcilive.o:  mcilive.c
        cc -c $(COPTS) mcilive.c

#*****************************************************************************/
#* The mciwdmg program inputs the mcilive network datafile and a damage      */
#* vector for MCI and LEC assets. It outputs a completed dataprep file for   */
#* input into LECAM.                                                         */
#*****************************************************************************/
mciwdmg:  mciwdmg.o fo.o region.o traffic.o selecteo.o /user/gretchen/waglib/waglib.o
        cc $(COPTS) mciwdmg.o fo.o region.o traffic.o selecteo.o /user/gretchen/waglib/waglib.o -lm -o mciwdmg

mciwdmg.o:  mciwdmg.c
        cc -c $(COPTS) mciwdmg.c

#*****************************************************************************/
#* The sprlive program performs calculations on the live Sprint network and  */
#* produces a live intermediate file for damage by sprwdmg.                  */
#*****************************************************************************/
sprlive:  sprlive.o traffic.o poisson.o phdiv.o loadlivekey.o
        cc $(COPTS) sprlive.o traffic.o poisson.o phdiv.o loadlivekey.o -lm -o sprlive

sprlive.o:  sprlive.c
        cc -c $(COPTS) sprlive.c

#*****************************************************************************/
#* The sprwdmg program inputs the sprlive network datafile and a damage      */
#* vector for Sprint and LEC assets. It outputs a completed dataprep file    */
#* for input into LECAM.                                                     */
#*****************************************************************************/
sprwdmg:  sprwdmg.o fo.o region.o traffic.o selecteo.o /user/gretchen/waglib/waglib.o
        cc $(COPTS) sprwdmg.o fo.o region.o traffic.o selecteo.o /user/gretchen/waglib/waglib.o -lm -o sprwdmg

sprwdmg.o:  sprwdmg.c
        cc -c $(COPTS) sprwdmg.c
```

List of Acronyms

AT	access tandem
AT&T	American Telephone & Telegraph
att	American Telephone & Telegraph
EO	end office
HU	High Usage
IEC	Inter-Exchange Carrier
LEC	Local Exchange Carrier
MCI	MCI Telecommunication Corporation
mci	MCI Telecommunication Corporation
NCAM	Network Connectivity Analysis Model
NCS	National Communication System
NLP	National Level NS/EP Telecommunications Program
NS/EP	National Security and Emergency Preparedness
NT	National Communications System (OMNCS) Office of Technology and Standards
OMNCS	Office of the Manager, National Communication System
POP	point of presence
PSN	Public Switched Network
QTCM	Queuing Traffic Congestion Model
SPR	Sprint
spr	Sprint
SW	switch
TAMI	Traffic Analysis by Method of Iteration
TG	Trunk Group

List of References

1. TAMI Model Programmer's Manual Volume I:, National Communications System, June 1995.

2. QTCM Software Documentation, Volume I: Programmer's Manual, National Communications System, November 1990.

3. Network Analysis Sensitivity Report, National Communications System, March 1994.

4. Network Analysis Report, National Communications System, June 1994.

5. Infrastructure Damage Assessment/Communication Assessment Model, Programmer's Manual, National Communications System, October 1990.

6. Network-Level EMP Effects Evaluation On The Primary PSN Toll-Level Networks, National Communications System, June 1993.

7. Network Congestion Analysis Report, National Communications System, November 1992.